Algoritam automatskog planiranja STRIPS u računalnim igrama

Jocković, Denis

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Denis Jocković

ALGORITAM AUTOMATSKOG PLANIRANJA STRIPS U RAČUNALNIM IGRAMA

DIPLOMSKI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Denis Jocković

Matični broj: 45938/17–R

Studij: Baze podataka i baze znanja

ALGORITAM AUTOMATSKOG PLANIRANJA STRIPS U RAČUNALNIM IGRAMA

DIPLOMSKI RAD

Mentor:

Izv. prof. dr. sc. Markus Schatten

Varaždin, rujan 2022.

Denis Jocković

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Tema rada je definiranje i prikaz primjene STRIPS algoritma. Teorijski dio rada započinje teorijskim objašnjenjem STRIPS algoritma i završava opisima jednostavnih primjera STRIPS algoritma. Nakon opisa primjera navedeni su bitni dijelovi aplikacije koji su se koristili u izradi algoritma. Završetkom opisa aplikacija korištenih u izradi programa započinje opis samog algoritma koji se u ovom radu koristi za izradu igre. U tom dijelu naveden je i opisan kôd korišten za izradu algoritma, kôd za povezivanje algoritma s gotovim strojem igara i kôd u korištenom gotovom stroju igara. Završno, u radu se može pronaći zaključak u kojem je naveden osvrt na algoritam i razmatranje o njegovoj koristi u stvarnom svijetu.

Ključne riječi: STRIPS; algoritam; računalna igra; prolog; java; Unreal Engine; avatar;

Sadržaj

1. Uvod

Industrija igara postaje sve profitabilnija i utjecajnija na svjetskoj sceni. Sve više ljudi igra igre na mobilnim uređajima i osobnim računalima. Zato se sve više poduzeća odlučuje za izradu igrara. Trendovi u industriji igara često se mjenjaju iako ne uvijek na bolje. Trendovi se mijenjaju kako bi poduzeća mogla maksimizirati profit. To se može vidjeti u brojnim izmjenama načina za monetizacije igrara. U početku su se igre naplaćivale te je dodatni sadržaj bio besplatan. Međutim, problem s takvim načinom monetizacije je u činjenici da je igru moguće naplatiti samo jednom. Iz tog razloga je velik broj poduzeća napustio takav sistem te je stvoren novi trend - *lootboxi*. Dosta dugo su *lootboxi* imali svoje mjesto u velikom broju igri, međutim, ni ovakav način monetizacije nije savršen i nosi sa sobom određene probleme. Primarno su neke države u *lootboxima* vidjele oblik klađenja te ih kao takve zabranile. Sljedeći je problem to što određeni broj igrača nije htio koristiti *lootboxe* zato što nisu htjeli trošiti novac, ako nebi bili sigurni da će zauzvrat dobiti nešto vrijedno novaca. Tako je osmišljen idući, i trenutni trend - *battle passovi*. Sistem *battle passa* je takav da se igranjem igre ili prelaskom misija dobijaju bodovi te se otključavaju razne nagrade koje se nalaze u *battle passu*. Neke od nagrada su besplatne, a većina, i to obično one najbolje, mogu se dobiti samo ako se uz otključavanje te nagrade dobije i dovoljan broj bodova. S takvim sistemom se slaže većina igrača jer se točno zna na što pojedinac troši, a igre koje imaju takav sustav obično su free-to-play.

Promijene u načinima monetizacije dovele su i do promijena u vrstama igara. Naime, dok su se igre plaćale, one su primarno bile single-player igre u kojima je naglasak bio na priči i gameplayu. *Lootboxi* i *battle passovi* ne mogu se dobro monetizirati u takvim igrama pa su se zato poduzeća fokusirala na izradu multi-player igri. Takve igre su one u kojima su priča i gameplay manje bitni, a komunikacija među igračima je bitna te je mnogima važno izgledati dobro kako bi zadivili druge igrače.

Multi-player igre često imaju NPC-jeve koji moraju izvršiti niz koraka kako bi došli do nekog cilja. Često ti koraci nisu unaprijed određeni već ovise o izborima igrača. To znači da akcije koje NPC treba izvršavati ovise o stanju koje je promjenjivo jer na njega uzrokuju igrači. Za izradu umjetne inteligencije za takve NPC-jeve dobro je koristiti STRIPS algoritam.

2. Metode i tehnike rada

U izradi ovog rada primarno je korištena literatura s interneta. Prilikom odabira izvora posebno se pridavala pažnja na kredibilitet izvora. Iz tog je razloga značajni dio literature zbir znanstvenih akademskih radova. Osim toga, bez obzira što je većina izvora tražena na internetu, među izvorima se ne nalaze samo internetske stranice već i znanstveni radovi, članci i prezentacije sveučilišnog porijekla. U izvorima su se tražili i praktični primjeri kako bi čitatelj mogao jednostavno razumijeti najjednostavnije primjere korištenja STRIPS algoritma prije nego se prijeđe na primjer izrađen zajedno sa ovim radom.

Za izradu igre i algoritma korišteno je nekoliko programskih alata. Primarno su korišteni alati SWI-Prolog, Unreal Engine, IntelliJ IDEA i Notepad++. SWI-Prolog je potreban za povezivanje programskih jezika Prolog i Java pomoću biblioteke JPL. Unreal Engine je gotov stroj igara koji se koristio za izradu igre u kojoj je implementiran STRIPS algoritam. IntelliJ IDEA je integrirano razvojno okruženje za programiranje s programskim jezikom Java. Ono se koristilo za povezivanje Unreal Enginea s Prologom. Notepad++ je uređivač teksta koji se može koristiti za pisanje kôda. U ovom slučaju se koristio za pisanje kôda programskog jezika Prolog u kojem je implementiran STRIPS algoritam.

Osim aplikacija navedenih u prethodnom odlomku valja navesti još neke bitne biblioteke i programske okvire korištene u izradi aplikacije. Najvažnija je biblioteka JPL koja se koristi za povezivanje Jave i Prologa. Nakon toga bitan je programski okvir Spring Framework koji se koristio za izradu REST servisa koji su omogućili spajanje Unreal Enginea sa Prologom. Za kraj, bitno je navesti dodatak za Unreal Engine VaRest koji služi za rad s REST servisima u Unreal Engineu.

3. Razrada teme

U ovom dijelu rada prvo je prikazana teorija STRIPS algoritma i par njegovih osnovnih izvedbi. Nakon toga se može vidjeti u kakvim situacijama se sve može koristiti STRIPS algoritam. Time je zaključen teorijski dio ovog rada. Slijedi prikaz osnovnih alata korištenih za izradu igre koja je temeljni dio ovog rada. Završetkom opisa osnovnih alata opisuju se i objašnjavaju najvažniji dijelovi Unreal Engina za ovaj projekt. Na kraju razrade detaljno se objašnjava izrada STRIPS algoritma za koncept igre koji je inspiriran Avatarom.

3.1. STRIPS algoritam

Stanford Research Institute Problem Solver (STRIPS algoritam) je algoritam koji služi za pronalaženje sekvence operatora pomoću kojih se neki početni model svijeta može transformirati u neki u kojem je određena formula istinita.[1] Prilikom korištenja STRIPS algoritma potrebno je opisati svijet.[2] To znači da je potrebno opisati početno stanje, završno stanje i pravila za prelazak iz jednog stanja u drugo.

Neki od problema koji se mogu rješiti sa STRIPS algoritmom su problem Rubikove kocke, igranje Starcrafta i upravljanje robotom u Shakeyevom svijetu.[2] U ovom radu opisan je problem slaganja blokova i problem majmuna i banane. U drugom dijelu rada opisan je i STRIPS algoritam koji služi za upravljanje NPC-jem koji mora skupljati različite stvari kako bi mogao evoluirati.

3.1.1. STRIPS algoritam kao algoritam planiranja

Kako bi se opisalo zašto je STRIPS algoritam algoritam planiranja, prvo je potrebno definirati što je to stanje. Početno i ciljno stanje važni su dijelovi STRIPS algoritma. Iz tog je razloga u ovom dijelu opisano na što se odnosi riječ stanje u kontekstu STRIPS algoritma. Početno ili završno stanje potpuni je opis svijeta koje se koristi prilikom rješavanja problema. Konkretno, početno stanje je stanje u kojem se svijet nalazi kada se započne rješavanje problema, a završno stanje je ono stanje u kojem se svijet treba nalaziti kada je problem riješen.[3]

Ovom prilikom valja opisati koncepte operatora i rješenja. Operatori su akcije koje transformiraju svijet iz jednog stanja u drugo. Rješenje je put kojim se dolazi iz početnog stanja u završno stanje. Vezano uz prethodna dva pojma, planiranje je pojam koji označava odabir uzastopnih akcija. Njihovim se izvođenjem dobije željeno ciljno stanje. Na isti način se može definirati i pretraživanje iako ipak postoje određene bitne razlike između pretraživanja i planiranja koje su navedene u nastavku:[3]

- Planiranje se oslanja na faktoriziranu zastupnjenost stanja i ciljeva koja se dobije pomoću određenih svojstava
- Mogućnosti i učinci operatora su objašnjeni u svojstvima pomoću kojih se dobije prethodno navedena faktorizirana zastupljenost

- Pomoću planiranja se mogu zaključiti posljedice određenih akcija
- Planiranjem se problemi mogu podijeliti u potprobleme čija se rješenja mogu otkriti, a samim time se i pronalazi rješenje problema
- Kod algoritama planiranja u obzir se moraju uzeti odgovori na dodatna pitanja koja se ne postavljaju kod algoritama pretraživanja:[3]
  - Može li se određena akcija izvršiti u trenutnom svijetu
  - Koje će posljedice na svijet određena akcija imati
- Kod planiranja se pretpostavlja da ako nešto nije navedeno da se mijenja akcijom, onda se obavezno ne može mijenjati

Iz navedenih se činjenica može vidjeti da je za uspješno izvođenje algoritama planiranja potrebno imati dobro znanje o svijeti te je to znanje potrebno izrazito točno prenijeti u algoritam. To u stvarnosti nije uvijek jednostavno jer je stvarni svijet kompleksan i izrazito dinamičan. Zbog tog se razloga algoritmi često pojednostavljuju, a za svijet se pretpostavlja da je statičan i da će izvođenje neke akcije uvijek imati točno određeni efekt na svijet.[3] Svijet naravno ne funkcionira na taj način, no bez takve naizgled jednostavne pretpostavke ne bi bilo moguće izvršiti algoritme planiranja jer bi bili previše kompleksni, a s time i značajno manje točnii.

Sada kada je čitatelj upoznat sa generalnim načinom rada algoritama pretraživanja može se prijeći na objašnjenje načina rada iščekivanog STRIPS algoritma. U STRIPS algoritmu su stanja i ciljevi prikazani kao skupovi literala. Njegove su akcije prikazane pomoću operatora koji imaju sljedeća svojstva:[3]

- Svaki operator ima naziv koji odgovara akciji
- Svaki operator ima određene uvjete koji moraju biti ispunjeni kako bi se izvršila akcija koju on predstavlja. Definirani uvjeti mogu biti prikazani kao skup stanja koji općenito mora biti istinit
- Svaki operator nekako utječe, što znači da je za svakog operatora potrebno odrediti način na koji će izvođenje akcije koju on predstavlja utjecati na svijet To se može prikazati kao način na koji će se trenutni skup stanja svijeta promijeniti

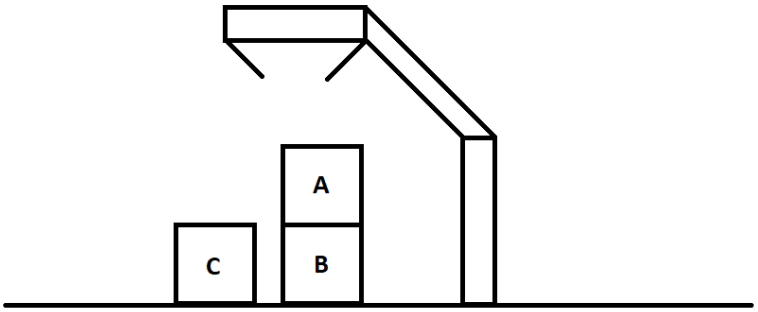
3.1.2. Jednostavni primjeri STRIPS algoritma

Nakon postavljenih konceptualnih teorijskih temelja slijede specifični primjeri algoritma za bolje shvaćanje opisane teorije. Primjeri su prikazani na način da je naveden kôd koji prikazuje na koji su način oni implementirani, a svaki isječak kôda je dodatno pojašnjen. Prikazani primjeri su problem slaganja blokova i problem majmuna i banane.

3.1.2.1. Problem slaganja blokova

Problem slaganja blokova česti je problem kojeg se koristi pri objašnjavanju STRIPS algoritma. Takav se problem nalazi u vrlo jednostavnom svijetu koji se sastoji od robotske ruke koja može biti prazna ili imati blok (paket ili kocka) i blokova koji mogu biti u robotskoj ruci, na drugom bloku ili na stolu. Dodatna pravila su da ruka može u određenom trenutku držati samo jedan blok te da može pokupiti samo one blokove koji nemaju druge blokove na sebi. Ruka može uvijek ostaviti blok kojeg drži na stolu ili na bilo kojem bloku koji nema blokove na sebi.

Kao što je već rečeno svaki STRIPS algoritam treba početno i ciljno stanje i operatore kojima se može doći iz početnog u ciljno stanje. U ovom slučaju, početno stanje govori o tome je li ruka prazna ili drži blok i gdje se svaki blok nalazi. Primjerice stanje [*rukaPrazna, na(A,B), naStolu(C), naStolu(B), vrhPrazan(C), vrhPrazan(A)*] definira da je početno stanje svijeta takvo da je robotska ruka prazna, da se blokovi B i C nalaze na stolu te da se blok A nalazi na bloku B. Prikaz opisanog može se jednostavnije predočiti slikom 1.



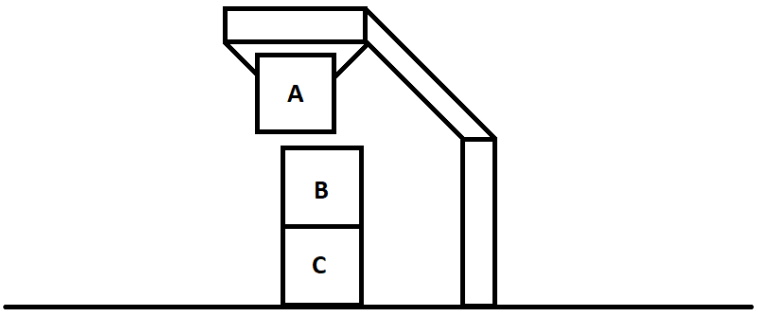
Slika 1: Početno stanje svijeta kod problema slaganja blokova

To je samo jedno od brojnih drugih mogućih početnih stanja opisanog svijeta. Neko drugo početno stanje moglo bi imati drugi broj blokova, drugačiji razmještaj blokova itd. U nekim početnim stanjima robotska ruka može imati blok pa i to čini dodatnu mogućnost razlike početnih stanja.

Nakon što je početno stanje definirano, vrijeme je da se definira ciljno stanje. Za ciljno stanje je važno da ima iste blokove kao i početno stanje jer bi inače bilo nemoguće doći do rješenja. U ciljnom stanju blokovi mogu biti razmješteni na bilo koji način, a robotska ruka može biti puna ili prazna.

Za ovaj primjer ciljno stanje je sljedeće: [*naStolu(C), vrhPrazan(B), na(B,C), uRuci(A)*]. Može se primjetiti da u ciljnom stanju robotska ruka nije prazna, ali je broj blokova i njihov naziv

jednak onome u početnom stanju. Na sljedećoj slici se može vidjeti kako izgleda ciljno stanje svijeta.

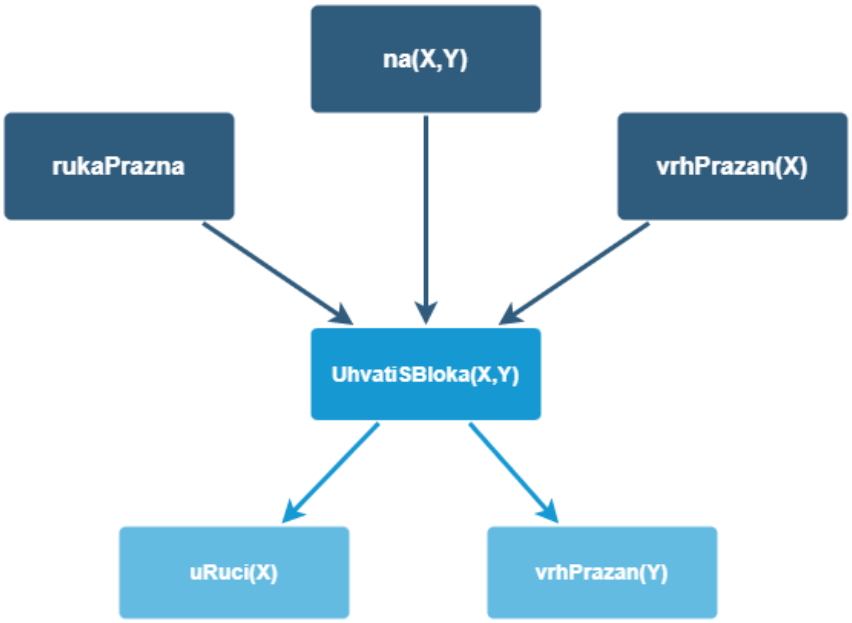


Slika 2: Ciljno stanje svijeta kod problema slaganja blokova (Izvor: autorova izrada)

Prema[3] moguće je još više pojednostaviti problem slaganja blokova. Primjerice, moguće je izuzeti ograničenja vezana za premještanje blokova. U tom slučaju ne bi bilo potrebe za robotskom rukom jer bi se blokovi mogli premještati bez ikakvih ograničenja. Međutim, takav problem bi vjerojatno bio previše trivijalan i u njemu se ne bi mogla prikazati korist i prednost STRIPS algoritma.

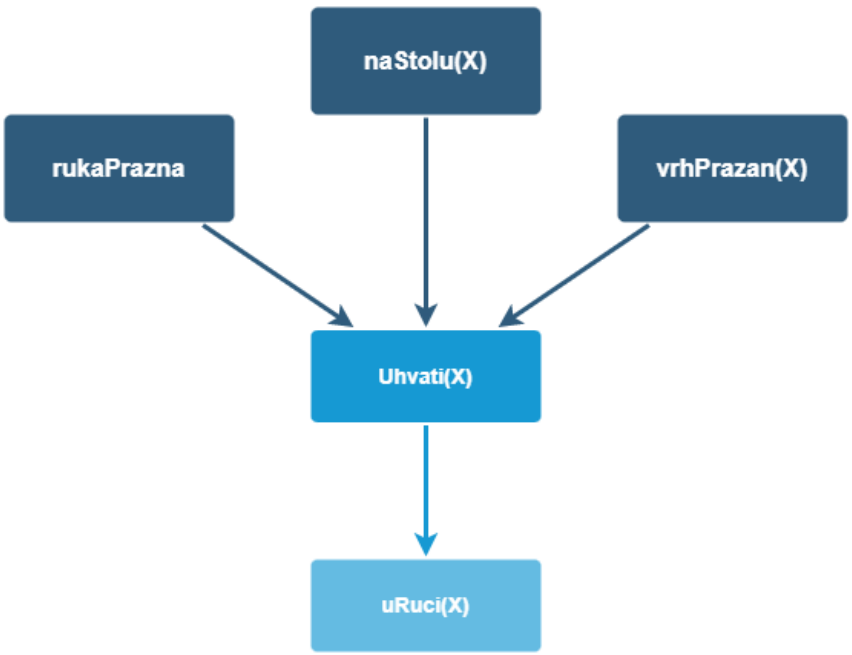
Definiranim početnim i ciljnim stanjem može se prijeći na akcije koje se mogu izvršiti u svijetu. Navedene akcije će se većinom odnositi na dizanje i spuštanje blokova pomoću robotske ruke. U nastavku su navedene i objašnjene sve akcije koje postoje u svijetu:

- *UhvatiSBloka(X, Y)* je akcija čijim izvršavanjem robotska ruka uzima blok X koji se nalazi na bloku Y[4]
  - Kako bi se navedena akcija mogla izvršiti sljedeći uvjeti moraju biti ispunjeni: *rukaPrazna, na(X,Y), vrhPrazan(X)*. Spomenuti uvjeti moraju biti ispunjeni jer robotska ruka može držati samo jedan blok u određenom trenutku, u akciji je definirano da se blok X nalazi na bloku Y, a bilo koji blok se može uhvatiti samo ako je na vrhu
  - Stanje svijeta će se izmjeniti na način da ruka više neće biti prazna već će imati blok X *rukaPrazna -> uRuci(X)*, blok X više neće biti na bloku Y i na vrhu zato što je u ruci: *na(X,Y), vrhPrazan(X) -> vrhPrazan(Y)*



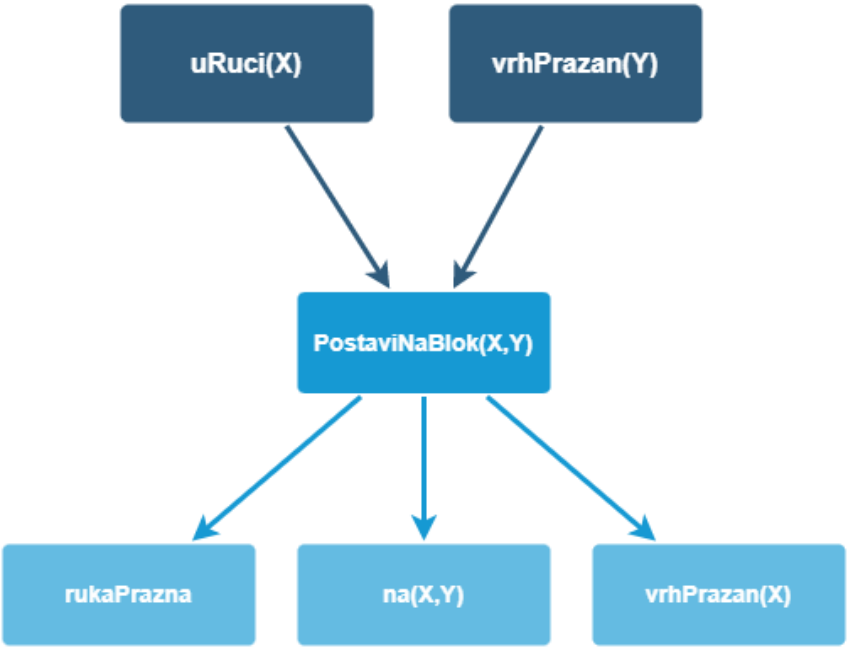
Slika 3: Grafički prikaz funkcije UhvatiSBloka (Izvor: autorova izrada, prema [5])

- *Uhvati(X)* je akcija čijim izvršenjem robotska ruka uzima blok X koji se nalazi na stolu[4]
  - Kako bi se navedena akcija izvršila sljedeći uvjeti moraju biti ispunjeni: *rukaPrazna, naStolu(X), vrhPrazan(X)*. Blok mora biti na stolu zato što akcija *Uhvati(X)* zahtjeva da je blok na stolu za razliku od akcije *UhvatiSBloka(X, Y)* koja zahtjeva da blok bude na drugom bloku
  - Stanje svijeta će se izmjeniti na način da ruka više neće biti prazna već će imati blok X *rukaPrazna -> uRuci(X)*, a blok X više neće biti na stolu i na vrhu zato što je u ruci *naStolu(X), vrhPrazan(X) -> ništa*



Slika 4: Grafički prikaz funkcije Uhvati (Izvor: autorova izrada, prema [5])

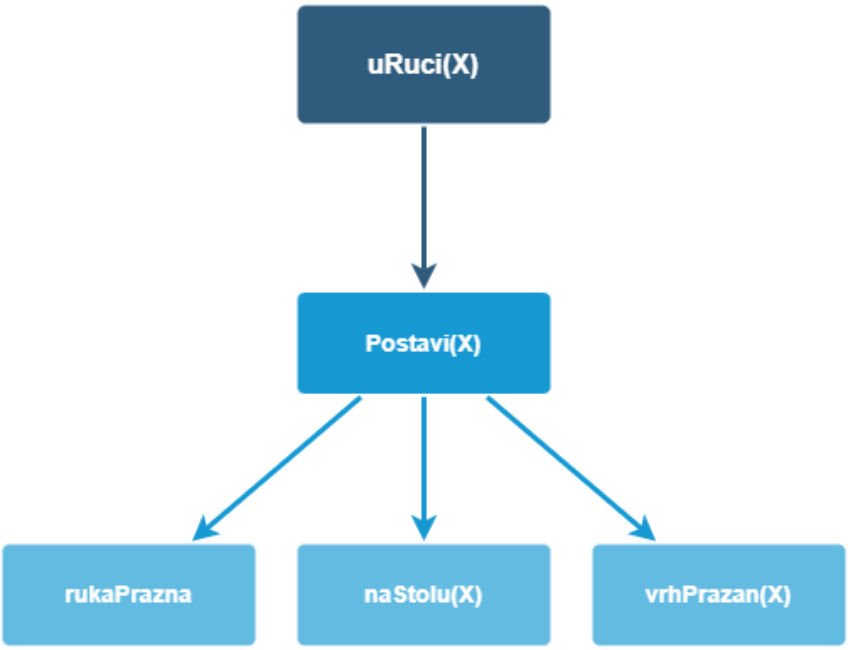
- *PostaviNaBlok(X, Y)* je akcija čijim izvršenjem robotska ruka stavlja blok X na blok Y[4]
  - Kako bi se navedena akcija izvršila sljedeći uvjeti moraju biti ispunjeni: *uRuci(X), vrhPrazan(Y)*. Spomenuti uvjeti moraju biti ispunjeni jer robotska ruka mora imati određeni blok kako bi ga mogla spustiti i blok se može spustiti samo na blok koji nema neki blok iznad sebe
  - Stanje svijeta će se izmjeniti na način da će ruka postati prazna *uRuci(X) -> rukaPrazna* i blok Y više neće biti na vrhu nego će blok X i biti na bloku Y i na vrhu: *vrhPrazan(Y) -> na(X,Y), vrhPrazan(X)*



Slika 5: Grafički prikaz funkcije PostaviNaBlok (Izvor: autorova izrada, prema [5])

- *Postavi(X)* je akcija čijim izvršenjem robotska ruka stavlja blok X na stol[4]
  - Kako bi se navedena akcija izvršila uvjet *uRuci(X)* mora biti ispunjen.
  - Stanje svijeta će se izmjeniti na način da će ruka postati prazna *uRuci(X) -> rukaPrazna* i da će blok X biti na stolu te da nad sobom neće imati drugi blok *ništa -> naStolu(X), vrhPrazan(X)*

Zanimljivo je da u ovom svijetu svaka akcija ima svoju "suprotnu akciju". Naime, akcija *UhvatiSBloka(X, Y)* ima suprotan rezultat od akcije *PostaviNaBlok(X, Y)*. Isto vrijedi i za akcije *Uhvati(X)* i *Postavi(X)*



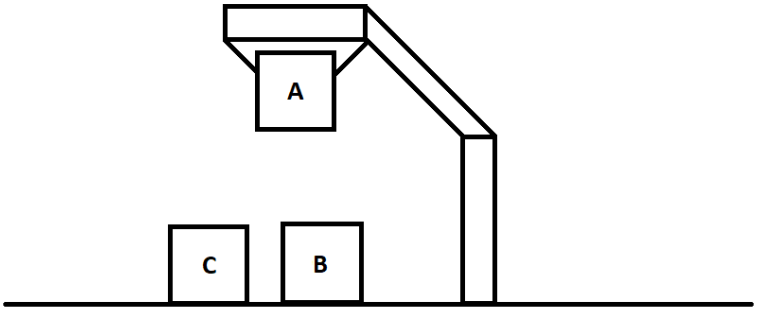
Slika 6: Grafički prikaz funkcije Postavi (Izvor: autorova izrada, prema [5])

3.1.2.2. Rješavanje primjera problema slaganja blokova STRIPS algoritmom

Opisani svijet problema slaganja blokova stvara potpuni kontekst za razumijevanje pa se način rješavanja primjera iz prethodnog poglavlja sada može predstaviti. Podsjetimo se, taj primer traži pretvaranje početnog stanja [*rukaPrazna, na(A,B), naStolu(C), naStolu(B), vrhPrazan(C), vrhPrazan(A)*] u ciljno stanje [*naStolu(C), vrhPrazan(B), na(B,C), uRuci(A)*]. Tok rješavanja teče redom na način da se gleda koja su stanja ispunjena, a za koja je stanja potrebno poduzeti određenu akciju. Prvi cilj *naStolu(C)* je već ispunjen u početnom stanju. To znači da se za ispunjavanje tog cilja ne mora izvršiti niti jedna akcija.

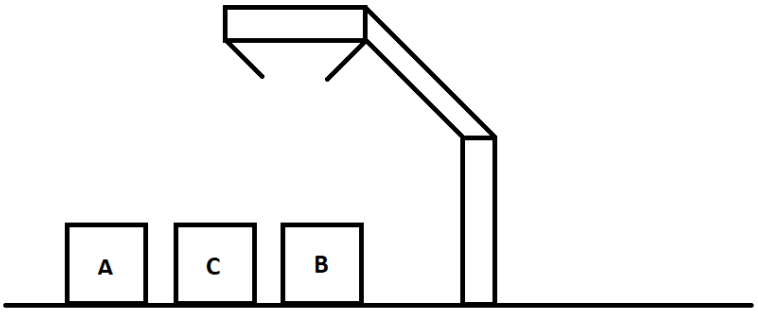
Drugi cilj *vrhPrazan(B)* ne vrijedi u trenutnom, početnom stanju stoga je potrebno nešto napraviti kako bi se taj cilj ispunio. S obzirom da *vrhPrazan(B)* ne vrijedi, zato što vrijedi *na(A,B)*, potrebno je podignuti blok A pomoću robotske ruke. To se radi akcijom *UhvatiSBloka(A, B)*. Kako bi navedenu akciju bilo moguće izvršiti određena stanja moraju trenutno biti istinita. Ruka mora biti prazna (*rukaPrazna*), blok A mora biti na bloku B (*na(A,B)*) i blok A mora biti na vrhu (*na(A,B)*). S obzirom da je to sve istinito (vidi slika 1), akcija se izvršava te se iz početnog stanja prelazi u stanje [*uRuci(A), naStolu(C), naStolu(B), vrhPrazan(C), vrhPrazan(B)*]. To je stanje vizualno predočeno na slici 7.

Idući cilj koji mora biti ispunjen u ciljnom stanju je *na(B,C)*. Taj cilj ne vrijedi u trenutnom stanju pa je potrebno izvršiti akciju koja će izmjeniti trenutno stanje tako da taj cilj vrijedi. Akcija koju je potrebno izvršiti je *PostaviNaBlok(B, C)*. Kako bi tu akciju bilo moguće izvršiti u trenutnom stanju moraju vrijediti sljedeća stanja *uRuci(B), vrhPrazan(C)* jer B mora biti u ruci da ga se može negdje postaviti i blok C ne smije imati blok na sebi kako bi se blok B na njega mogao postaviti. Sada je potrebno vidjeti jesu li ta dva uvjeta ispunjena. Odgovor je ne jer već prvi uvjeti *uRuci(B)* nije ispunjen u trenutnom stanju. To znači da je prije izvršavanja akcije *PostaviNaBlok(B, C)* potrebno izvršiti akciju *Uhvati(B)*. Kako bi tu akciju bilo moguće izvršiti uvjeti



Slika 7: Prva akcija kod rješavanja problema (Izvor: autorova izrada)

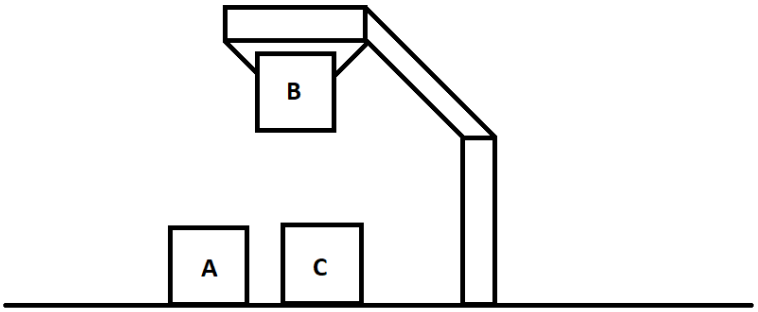
koji moraju biti ispunjeni su *rukaPrazna, naStolu(B), vrhPrazan(B)*. Niti ovaj put nisu svi uvjeti ispunjeni jer ne vrijedi *rukaPrazna* s obzirom da je blok A trenutno u ruci. Kako bi se taj uvjet ispunio potrebno je izvršiti akciju *Postavi(A)* da se blok A postavi na stol i da se ruka oslobodi. Kako bi se ta akcija ispunila *uRuci(A)* mora biti istinito. S obzirom da je jedini uvjet potreban za ispunjavanje te akcije ispunjen akcija se može izvršiti. Izvršenjem te akcije trenutno stanje se mjenja u sljedeće stanje: [*rukaPrazna, naStolu(A), naStolu(C), naStolu(B), vrhPrazan(C), vrhPrazan(B), vrhPrazan(A)*]. Novo se stanje može vidjeti na slici 8.



Slika 8: Druga akcija kod rješavanja problema (Izvor: autorova izrada)

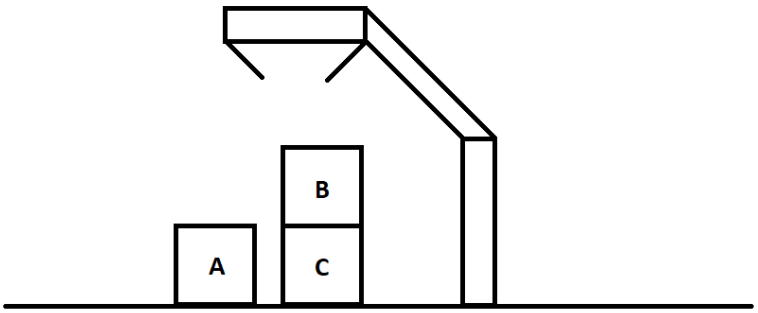
Promjenom stanja uzrokovanom prethodnom akcijom možda je omogućeno izvršavanje akcije *Uhvati(B)*. Naime, kada prilikom izvršavanja algoritma dođe do situacije u kojoj je potrebno izvršiti više akcija kako bi se jedan od ciljeva u ciljnom stanju obistinio, onda se sve te akcije moraju izvršiti prije nego se prijeđe na ispunjavanje novog cilja u nizu ciljeva ciljnog stanja. Osim toga, akcije se izvršavaju na način da se akcija koju je algoritam zadnju spomenuo izvršava najprije. Iz tog razloga je sada potrebno izvršiti akciju *Uhvati(B)* ukoliko su za to ispunjeni uvjeti.

Za izvršavanje akcije *Uhvati(B)* izvršeni su svi uvjeti što znači da se ona može izvršiti. Izvršavanjem te akcije stanje svijeta postaje [*uRuci(B), naStolu(A), naStolu(C), vrhPrazan(C), vrhPrazan(A)*]. Na slici 9 se može vidjeti stanje svijeta u ovom trenutku.



Slika 9: Treća akcija kod rješavanja problema (Izvor: autorova izrada)

Nakon izvršavanja akcije *Uhvati(B)* potrebno je odrediti je li moguće izvršiti akciju *PostaviNaBlok(B, C)*. Tu je akciju zaista moguće izvršiti jer su ispunjeni svi uvjeti za njezino izvršavanje. Nakon izvršavanja te akcije stanje je [*rukaPrazna, naStolu(C), vrhPrazan(B), na(B,C), naStolu(A), vrhPrazan(A)*].Trenutno stanje može se vidjeti na slici 10.



Slika 10: Četvrta akcija kod rješavanja problema (Izvor: autorova izrada)

S obzirom da je cilj *na(B,C)* ispunjen, može se krenuti s ispunjavanje idućeg cilja. Idući je cilj *uRuci(A)*. Kako bi se taj cilj ispunio robotska ruka mora držati blok A što znači da mora biti izvršena akcija *Uhvati(A)* s obzirom da je A na stolu. Kako bi se ta akcija mogla izvršiti moraju biti ispunjeni uvjeti *rukaPrazna*, *naStolu(A)* i *vrhPrazan(A)*. S obzirom da su svi ti uvjeti ispunjeni (vidi slika 10) može se krenuti s izvršavanjem akcije. Nakon izvršavanja akcije stanje će biti [*naStolu(C), vrhPrazan(B), na(B,C), uRuci(A)*] što je ujedno i ciljno stanje. Sada kada je ciljno stanje postignutno može se vidjeti koje je akcije (i kojim redom) bilo potrebno izvršiti kako bi se od početnog stanja došlo do ciljno stanja:

1. *UhvatiSBloka(A, B)*
2. *Postavi(A)*
3. *Uhvati(B)*
4. *PostaviNaBlok(B, C)*
5. *Uhvati(A)*

Dakle, izvođenjem STRIPS algoritma u navedenom svijetu dolazi se do zaključka da je od početnog do ciljnog stanja moguće doći izvođenjem pet navedenih koraka točno definiranim redoslijedom. To je ujedno i najkraći put do ciljnog stanja što znači da do ciljnog stanja nije moguće doći u četiri ili manje koraka. Naravno, STRIPS algoritam uzima u obzir i druge kombinacije koraka koje mogu dovesti do rješenja, ali je logično uzeti ono koje ima najmanje koraka.

3.1.2.3. Programski kôd STRIPS algoritma za problem slaganja blokova

U ovom dijelu prikazan je i objašnjen programski kôd korišten za rješavanje problema slaganja blokova STRIPS algoritomom. Cilj ovog dijela je dodatno upoznavanje sa STRIPS algoritmom kako bi praktični dio ovog rada bio što jasniji. STRIPS algoritam koji će biti prikazan pisan je u programskom jeziku Prolog te preuzet i prilagođen prema izvoru na stranici swish.swi-prolog.org.[6]

Prva funkcija koja se navodi jest ista ona koju je potrebno zapisati u konzolu kako bi se dobio rezultat. Ta funkcija je solve/3 te se u nju zapisuju početno stanje, ciljno stanje i varijabla u koju će biti zapisani koraci kojima se od početnog dolazi do ciljnog stanja. Definicija te funkcije glasi

```
solve(Initial, Final, Plan) :- strips(Initial, Final, Plan).¶
```

Iduća funkcija koja se spominje je ona koja se poziva u funkciji solve/3, a to je strips/3. Funkcija strips/3 je definirana na način:

```
strips(Initial, Final, Plan) :- strips(Initial, Final, [Initial], Plan).
```

Pritom varijable označavaju iste elemente kao i kod prethodne funkcije. Sljedeća funkcija koja se spominje jest strips/4 koja je definirana na način:

```
strips(Initial, Final, Visited, Plan) :-
    deepening_strips(1, Initial, Final, Visited, Plan).
```

Ta funkcija poziva funkciju deepening_strips/5 koja je rekurzivna te služi za pozivanje funkcije bounded_strips/5 u kojoj se nalaze funkcije za provjeravanje mogućnosti izvršavanja akcija i funkcije za izvršavanje akcija. Navedena rekurzivna funkcija je definirana na sljedeći način:

```
deepening_strips(Bound, Initial, Final, Visited, Plan) :-
    bounded_strips(Bound, Initial, Final, Visited, Plan).
deepening_strips(Bound, Initial, Final, Visited, Plan) :-
    succ(Bound, Successor),
    deepening_strips(Successor, Initial, Final, Visited, Plan).
```

Funkcija bounded_strips/5 prvo izvršava funkciju succ/2 koja je sastavni dio osnovne funkcionalnosi prologa. Funkcija succ/2 je funkcija koja se brine da argumenti budu prirodni brojevi. Slijedi poziv funkcije action/2 koja služi kako bi se otkrilo jesu li svi preduvjeti za izvršavanje akcije ispunjeni. Nakon toga se poziva funkcija perform/3 koja služi za izvršavanje akcije. Nakon toga se poziva funkcija member/2 u kojoj se provjerava nalazi li se lista prvog argumenta u listi drugog argumenta. Ukoliko se ne nalazi izvršavanje se prekida, a inače se ponovno poziva ista funkcija. Prema tome, funkcija bounded_strips/5 također je rekurzivna. Funkcija ima programski kôd:

```
bounded_strips(_, Final, Final, _, []).
bounded_strips(Bound, Initial, Final, Visited, [Action|Actions]) :-
    succ(Predecessor, Bound),
    action(Initial, Action),
    perform(Initial, Action, Intermediate),
    \+ member(Intermediate, Visited),
    bounded_strips(Predecessor, Intermediate, Final,
        [Intermediate|Visited], Actions).
```

Funkcija action/2 provjerava: je li blok koji se treba pomaknuti zaista blok, može li se blok pomaknuti na odabrano mjesto te je li odabrano mjesto slobodno. Ukoliko su svi uvijeti istiniti, a pritom se blok ne nalazi na odabranom mjestu, funkcija će vratiti istinu. Navedena funkcija je definirana na način:

```
action(State, move(Block, Destination)) :-
    block(Block),
    \+ Block == Destination,
    free(State, Block),
    free(State, Destination).
```

Funkcija action/2 poziva funkciju free/2 koja provjerava postoji li u listi stanja onakvo stanje kakvo se želi postići izvršavanjem akcije. Provjerava se pomoću ugrađene funkcije member/2. Funkcija free/2 i povezana funkcija thing/1 definirane su na sljedeći način:

```
free(State, Thing) :-
    thing(Thing),
    \+ member(on(_, Thing), State).

thing(Block) :- block(Block).
thing(Place) :- place(Place).
```

Iduća funkcija koju je potrebno pobliže opisati je perform/3. Navedena funkcija služi za obavljanje željenih izmjena u listi stanja. Nakon izvođenja te funkcije iz liste određena stanja trebaju biti izbrisana, a druga dodana. Prvi argument te funkcije je postojeća lista, drugi argument je akcija koja se treba izvršiti, a treći je lista koja će nastati izvršavanjem akcije.

```
perform(Source, move(Block, Destination), Target) :-
    substitute(on(Block, _), Source, on(Block, Destination), Target).
```

Kao što se može vidjeti u gore prikazanom kôdu, funkcija perform/3 poziva funkciju substitute/4. Ta funkcija je rekurzivna i obavlja potrebne promjene liste. U nastavku se vidi programski kôd te funkcije:

```
substitute(_, [], _, []).
substitute(A, [A|As], B, [B|Bs]) :-
```

```
    substitute(A, As, B, Bs), !.
substitute(A, [X|As], B, [X|Bs]) :-
    substitute(A, As, B, Bs).
```

Funkcija substitute/4 zadnja je funkcija koja je potrebna kako bi STRIPS algoritam za ovaj jednostavan svijet bio funkcionalan. Ukratko, može se vidjeti da algoritam radi na način da prvo određuje jesu li svi uvjeti za potrebnu akciju i, ako jesu mijenja listu stanja prema definiranom načinu na koji se ona treba izmjeniti. Ovaj algoritam je naizgled jednostavan, ali za njegovu je implementaciju zapravo potrebno pozamašno znanje Prologa.

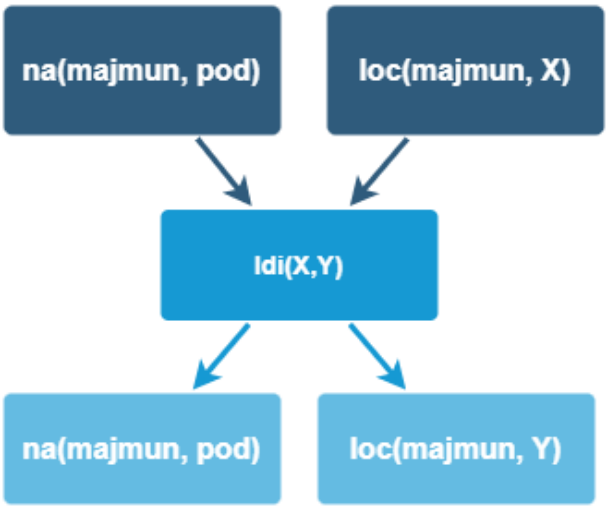
3.1.2.4. Problem majmuna i banane

Još jedan problem koji se može rješiti STRIPS algoritmom je problem majmuna i banane. Ukratko, to je svijet u kojem, između ostalog, postoje majmun i banana te majmun želi doći do banane kako bi je pojeo. U ovom slučaju algoritam treba dati niz akcija koje će dovesti majmuna do banane. Unutar svijeta ovog problema često postoji i stolica ili neki drugi objekt na kojeg se majmun treba popeti kako bi dosegnuo bananu koja obično visi na drvetu.

U ovom slučaju postoji više akcija koje majmun može izvršiti. Primjerice, majmun može hodati naokolo, može se popeti na stolicu, gurati stolicu i dohvaćati bananu. [7] U nastavku je svaka od akcija pobliže objašnjena.

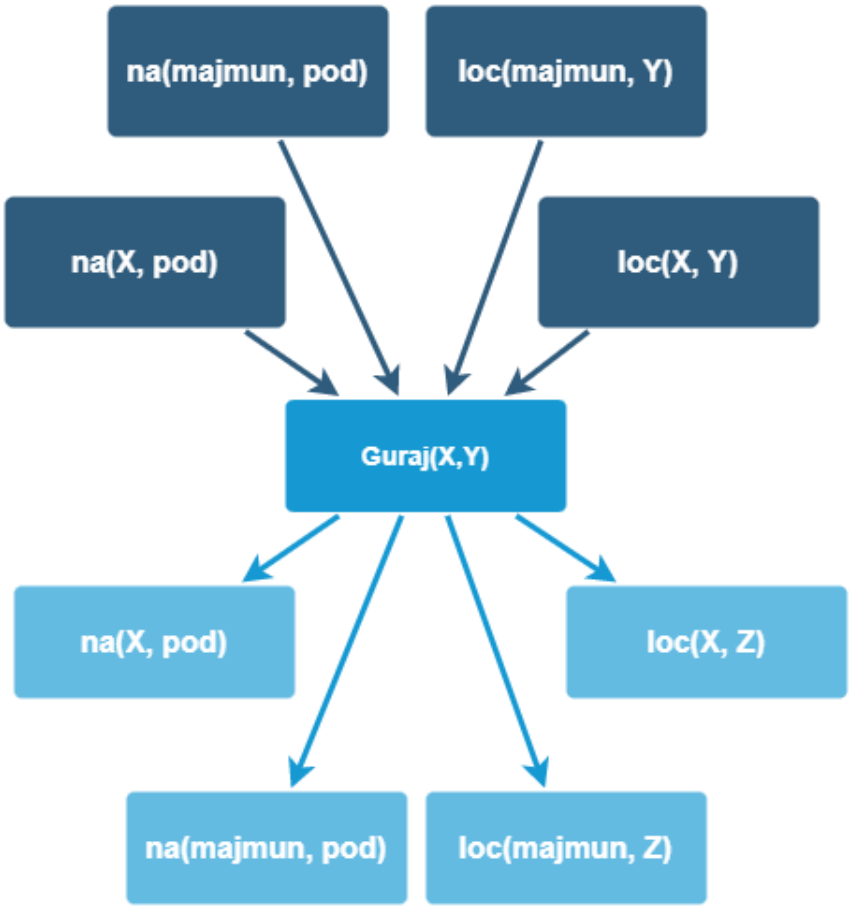
Ono što je prvo potrebno odrediti su početna i ciljna stanja. Početno stanje može biti bilo kakvo, ali se obično uzima u obzir da majmun nije na stolici i da stolica nije ispod drveta banana jer u tom slučaju problem postaje trivijalan. Ciljno stanje je bilo koje stanje u kojem majmun ima bananu.[7] Akcije koje majmun može poduzeti objašnjene su na sljedeći način:

- *Idi(X, Y)* je akcija koja služi za pomicanje majmuna s mjesta X na mjesto Y[7]
  - Kako bi se akcija izvršila majmun mora biti na podu (*na(majmun, pod)*) jer se ne može šetati naokolo, ako je na stolici. Majmun, također mora biti na mjestu X (*loc(majmun, X)*)
  - Stanja će se izmijeniti na način da majmun više neće biti na mjestu X nego na mjestu Y *loc(majmun, X) -> loc(majmun, Y)*



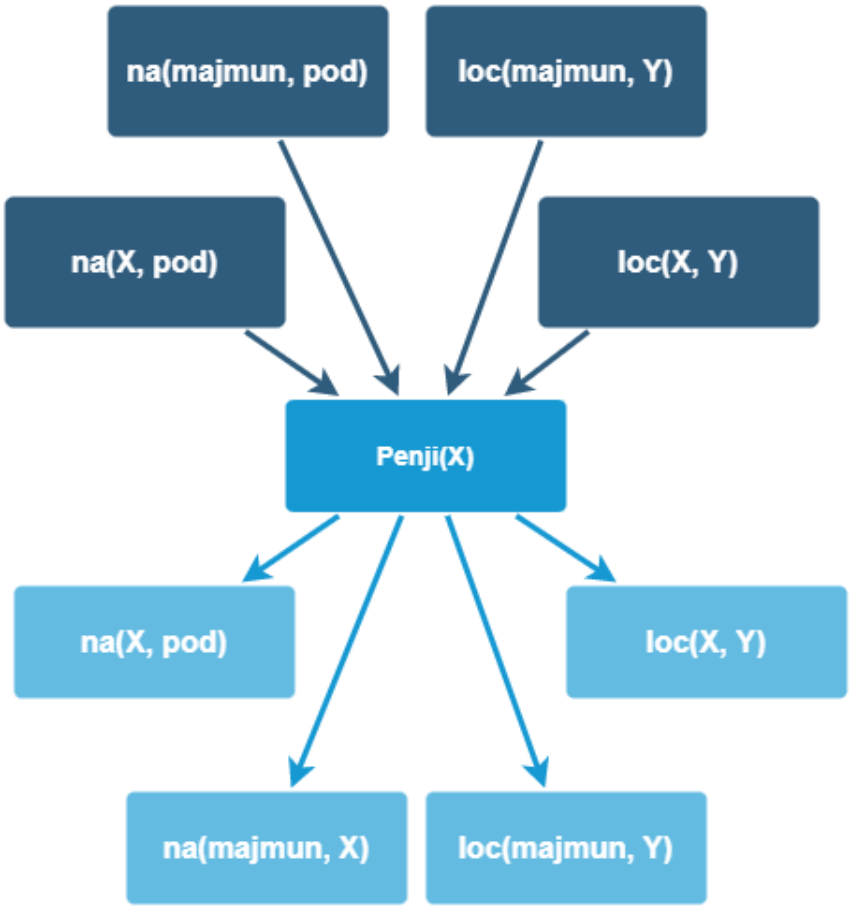
Slika 11: Grafički prikaz funkcije Idi (Izvor: autorova izrada, prema [5])

- *Guraj(X, Y, Z)* je akcija čijim izvršavanjem majmun gura predmet X s mjesta Y na mjesto Z[7]
  - Kako bi se navedena akcija izvršila sljedeći uvjeti moraju biti ispunjeni: *loc(majmun, Y), loc(X, Y), na(majmun, pod), na(X, pod)*. Majmun i predmet moraju biti na podu jer majmun može gurati stvari samo ako su na podu
  - Stanje svijeta će se izmjeniti na način da majmun i predmet X više neće biti na mjestu Y nego na mjestu Z *loc(X, Y), loc(majmun, Y) -> loc(X, Z), loc(majmun, Z)*
- *Penji(X)* je akcija kojom se majmun penje na predmet X. Obično se izvodi kada je stolica kod drveta.[7]
  - Kako bi se navedena akcija izvršila sljedeći uvjeti moraju biti ispunjeni: *loc(majmun, Y), loc(X, Y), na(majmun, pod), na(X, pod)*. Ti uvjeti moraju biti ispunjeni jer se majmun može popeti na predmet samo ako su na istom mjestu, samo na predmet koji je na tlu i ako je sam predmet na tlu
  - Stanje svijeta će se izmjeniti na način da majmun više neće biti na podu nego na predmetu X *na(majmun, pod) -> na(majmun, X)*



Slika 12: Grafički prikaz funkcije Guraj (Izvor: autorova izrada, prema [5])

- *Uhvati(X)* je akcija izvršavanjem koje majmun može dohvatiti predmet X. Koristi se za dohvaćanje banane s drveta[7]
  - Kako bi se akcija izvela potrebno je da majmun bude blizu drveta koje na sebi ima bananu, da majmun bude na stolici, a samim time i da je stolica blizu drveta: *loc(stolica, Y), na(majmun, stolica), loc(X, Y), posjed(X, drvo)*
  - Stanje svijeta će se izmjeniti na način da će majmun dobiti X sa drveta: *posjed(X, drvo) -> posjed(X, majmun)*



Slika 13: Grafički prikaz funkcije Penji (Izvor: autorova izrada, prema [5])

3.1.2.5. Programski kôd STRIPS algoritma za problem majmuna i banane

U ovom poglavlju prikazan je programski kôd rješavanja problema objašnjenog u prethodnom poglavlju. Prvi primjeri programskog kôda vezani su za akcije koje su objašnjene u prethodnom poglavlju. Programsko rješenje u ovom poglavlju preuzeto je sa stranice instituta za računalstvo i informacijske znanosti. [7] U sljedećem redu je kôd za akciju Idi:

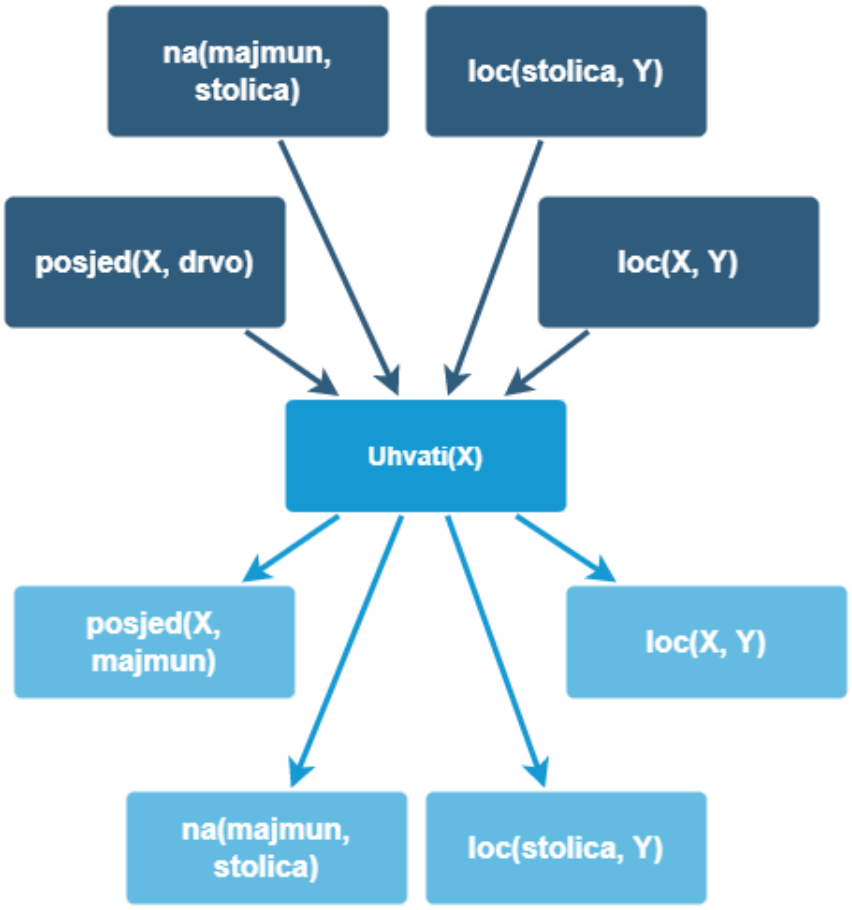
```
action(go(X,Y), [at(monkey,X), on(monkey,floor)],
    [at(monkey,X)], [at(monkey,Y)]).
```

Nakon toga slijedi akcija Guraj:

```
action(push(B,X,Y),
    [at(monkey,X), at(B,X), on(monkey,floor), on(B,floor)],
    [at(monkey,X), at(B,X)], [at(monkey,Y), at(B,Y)])
```

Akcija Penji ima sljedeći programski kôd:

```
action(climbon(B),
    [at(monkey,X), at(B,X), on(monkey,floor), on(B,floor)],
    [on(monkey,floor)], [on(monkey,B)]).
```



Slika 14: Grafički prikaz funkcije Uhvati (Izvor: autorova izrada, prema [5])

I zadnja akcija je Uhvati:

```
action(grab(B),
    [on(monkey,box), at(box,X), at(B,X), status(B,hanging)],
    [status(B,hanging)], [status(B,grabbed)]).
```

Osim akcija, u programskom rješenju su spomenute i funkcije koje služe za omogućavanje izvođenja STRIPS algoritma. Te funkcije (plan/3, plan/4) slične su funkcijama koje su korištene u objašnjavanju problema slaganja blokova (strips/3, strips/4) pa se njihovo objašnjenje može potražiti u tom poglavlju. U nastavku su prikazane navedene funkcije:

```
plan(State, Goal, Plan):-
    plan(State, Goal, [], Plan).
plan(State, Goal, Plan, Plan):-
    is_subset(Goal, State), nl,
    write_sol(Plan).
plan(State, Goal, Sofar, Plan):-
    action(A, Preconditions, Delete, Add),
    is_subset(Preconditions, State),
    \+ member(A, Sofar),
```

```
    delete_list(Delete, State, Remainder),
    append(Add, Remainder, NewState),
    plan(NewState, Goal, [A|Sofar], Plan).
```

Osim toga je u ovom programskom rješenju navedena i testna fukcija koja u sebi kao argument ima početno stanje, ciljno stanje i varijablu u koju će se zapisivati koraci za dolazak iz početnog u ciljno stanje. Ta funkcija je:

```
test1(Plan):-
    plan([on(monkey,floor),on(box,floor),at(monkey,loc1),at(box,loc2),
        at(bananas,loc3),status(bananas,hanging)],
    [status(bananas,grabbed)],
        Plan).
```

Može se vidjeti da su u početnom stanju te funkcije majmun i stolica na podu, također se vidi da su majmun, stolica i drvo na različitim mjestima te da su banane na drvetu. U završnom stanju majmun ima banane.

3.1.3. Metode za dolazak do rješenja u STRIPS algoritmu

U prethodnim poglavljima objašnjeno je što je STRIPS algoritam te su prikazani i objašnjeni primjeri STRIPS algoritma. U ovom dijelu objašnjeno je na koji način funkcionira navedeni algoritam. Funkcioniranje STRIPS algoritma može se prikazati pomoću grafa u kojem se nalaze sva trenutna stanja te se promatra na koji način će određena akcija dovesti do nekog drugog stanja. Ti grafovi nazivaju se grafovima planiranja.[2]

Nakon što se konstruira graf planiranja potrebno je slijedno pratiti njegove čvorove u određenom prolasku kako bi se pronašlo rješenje nekog problema. Grafovima se može prolaziti na tri načina: pretraživanjem u širinu, pretraživanjem u dubinu i A* pretraživanjem. Neki smatraju A* pretraživanje najboljim pristupom zato što se u prosjeku najbrže dođe do rješenja.[2]

Pretraživanje u širinu prvo pretraži svu djecu početnih stanja i kada provjeri svu djecu u toj razini, onda prelazi na sljedeću. Na taj način može prekinuti pretraživanje ukoliko je neki od čvorova djece rješenje. S obzirom da ovaj način provjerava svako dijete, pretraživanje u širinu može prilično dugo trajati. Također, što je graf širi, to pretraživanje traje duže. Dobra strana ovog pretraživanja je da će ono uvijek vratiti optimalno rješenje zato što će rješenje koje vrati uvijek biti na razini koja je najbliža korijenu grafa (stabla).[2]

Pretraživanje u dubinu radi na način da se gleda je li čvor na kojem se trenutno nalazi rješenje i ako nije onda se gleda dijete tog čvora i tako dok se može ići u dubinu. Dobra strana tog pretraživanja je da je obično brže nego pretraživanje po širini. Loša strana je da dobiveni rezultat ne mora nužno biti i optimalan.[2]

A* pretraživanje je heuristično pretraživanje. To znači da koristi formulu kako bi koliko je stanje u određenom čvoru slično ciljnom stanju. Na taj način A* pretraživanje može prioritizirati

one putanje koje daju veću šansu za otkrivanje ciljnog stanja. Taj algoritam obično daje najbrža rješenja jer se za pretraživanje koristi logika. Loša strana je da za razliku od pretraživanja u širinu postoji mogućnost da A* pretraživanje neće pronaći optimalno rješenje.

3.1.4. Izrada STRIPS algoritma za računalnu igru

Za potrebe demostracije STRIPS algoritma implementirana je jedna mehanika koja kao takva može samostalno postojati i djelovati unutar virtualnog svijeta video igre. Mehanika je jedan obično manji aspekt video igre koji se može odnositi na različite dijelove agoritama ili kompletne agoritme koji obavljaju određeni dio zadaće unutar video igre za pojedinu ili više svrha. Isto tako, postoji velika razlika između prototipa igre i njene završne verzije što se može i zaključiti iz slika zaslona u nastavku koji prikazuju rad prototipa s ugrađenim opisanim algoritmom. Lako se može primjetiti da se takav prototip uvelike razlikuje od oku poznatih igara na tržištu. Za jednostavnost razumijevanja koncepata koji su najvažniji za razmijevanje ovog rada sve opisano će se jednostavno oslovljavati kao igra.

Inspiracija za izradu algoritma bila je animirana serija Avatar: Posljednji gospodar vjetrova. Poanta igre je upoznavanje sa svim elementima (zrak, voda, zemlja i vatra) u svrhu evolucije u Avatara koji može kontrolirati sve elemente. Dizajnirani algoritam trebao bi kontrolirati NPC-ja na način da NPC traži stvari potrebne za svladavanje elemenata. Također, algoritam treba u potpunosti voditi NPC-ja na putu od nepoznavanja elemenata do evolucije u Avatara.

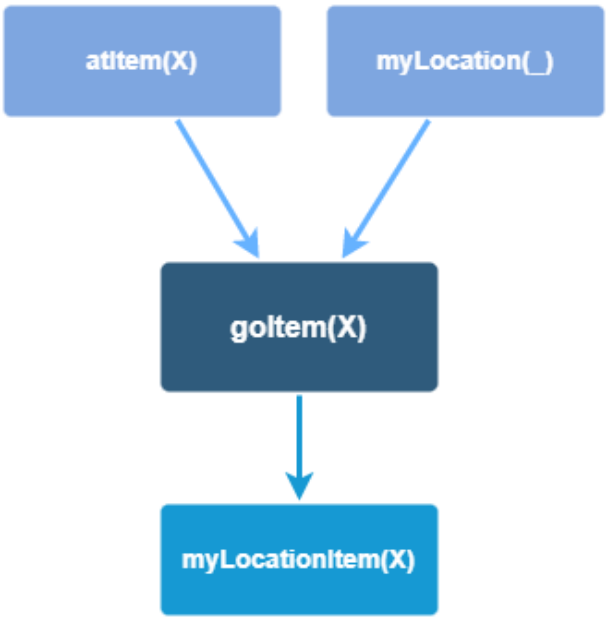
Najprije je potrebno objasniti na koji način se postaje Avatarom. U animiranoj seriji, osoba postaje Avatar po rođenju, međutim to nije praktično za demonstraciju STRIPS algoritma. Igra je dizajnirana na način da, za razliku od serije, svi mogu postati Avatari. U igri se elementi svladavaju na način da lik mora pronaći određenu stvar koja simbolizira specifični element (primjerice pero simbolizira zrak). Nakon toga moraju otići do oltara koji je vezan za element te na taj način "naučiti" element. Bitno je napomenuti da nije važno redom skupiti jednu stvar pa odmah otići na oltar već se mogu skupiti sve stvari ili njih nekoliko te onda otići na oltar. Nakon što su naučeni svi elementi, NPC i dalje nije Avatar. Kako bi konačno postao Avatar, NPC mora otići do posebnog oltara na kojem će evoluirati u Avatara.

Detaljnijim promatranjem pravila može se zamijetiti da za svaki od četiri elementa postoji samo jedna stvar koja pridonosi evoluciji. Na početku igre, NPC ne zna gdje se točno nalaze stvari koje treba skupiti niti gdje se nalaze oltari nego sve to trebaju pronaći. Učenje elemenata mora se odviti točno određenim redoslijedom zato što se smatra da su određeni elementi preduvijet za svladavanje drugih.

3.1.4.1. Stanja i akcije STRIPS algoritma za igru

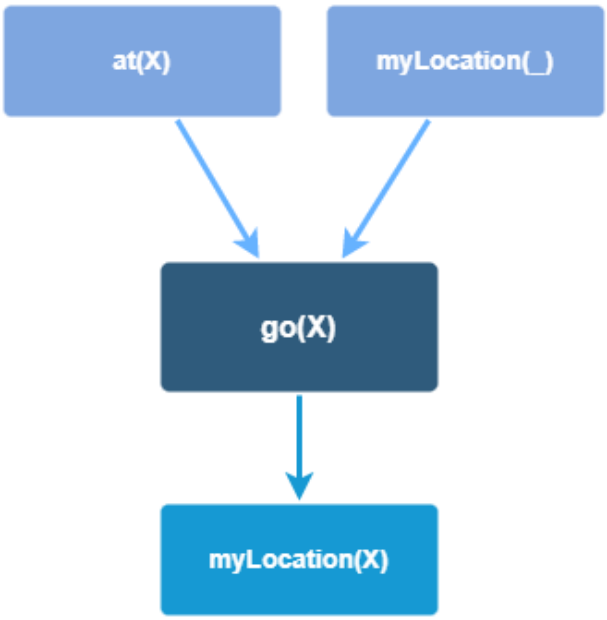
Sada kada su objašnjena pravila, može se prijeći na objašnjavanje početnih i ciljnih stanja te akcija koje se mogu poduzeti. U početnom stanju NPC ima početnu lokaciju i ništa drugo. U ciljnom je stanju NPC Avatar. Osim toga, postoji osam akcija koje NPC može učiniti, a u nastavku je objašnjena svaka od njih.

- *goItem(X)* je akcija koja govori NPC-ju da treba otići do određene stvari
  - Kako bi se navedena akcija izvršila NPC mora biti na bilo kojoj lokaciji na kojoj nije neka stvar *myLocation(_)* te mora znati da se stvar nalazi na lokaciji X *atItem(X)*
  - Stanja će se izmijeniti na način da NPC više neće biti na lokaciji na kojoj nema predmeta već će biti na lokaciji predmeta *atItem(X), myLocation(_) -> myLocationItem(X)*



Slika 15: Grafički prikaz funkcije goItem (Izvor: autorova izrada, prema [5])

- *go(X)* je akcija izvršavanjem koje NPC ide s mjesta na kojem nema stvari do drugog mjesta na kojoj nema stvari
  - Kako bi se navedena akcija izvršila NPC mora biti na bilo kojoj lokaciji na kojoj nije neka stvar *myLocation(_)* te mora znati da se nešto što nije stvar nalazi na lokaciji X *at(X)*
  - Stanja će se izmijeniti na način da NPC promijeniti lokaciju na kojoj se nalazi *at(X), myLocation(_) -> myLocation(X)*

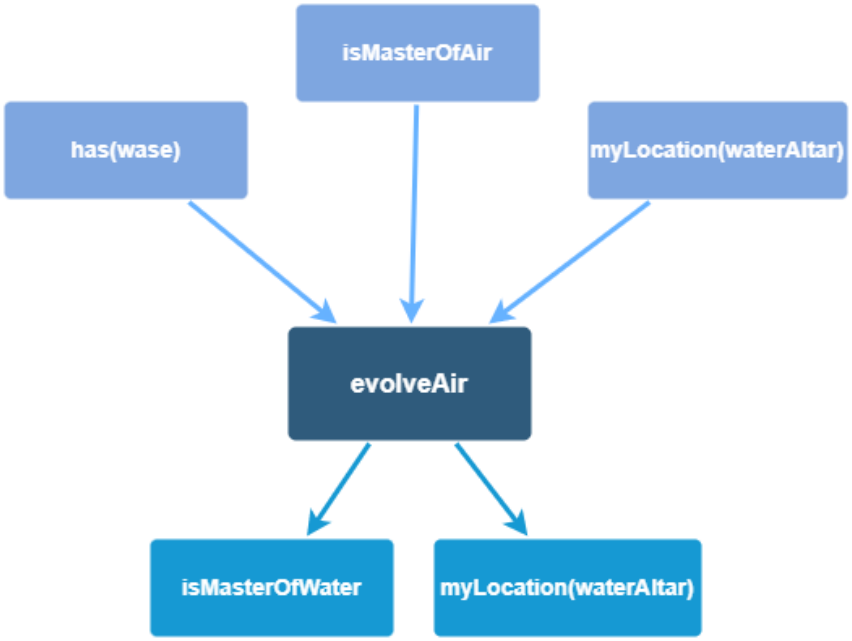


Slika 16: Grafički prikaz funkcije go (Izvor: autorova izrada, prema [5])

- *evolveAir* je akcija koja služi za učenje elementa zraka
  - Kako bi se navedena akcija izvršila sljedeći uvjeti moraju biti ispunjeni: *myLocation(airAltar), has(feathers)*. Ti uvjeti moraju biti ispunjeni jer se zrak može naučiti samo ako je NPC na oltaru zraka i ako ima stvar koja je potrebna kako bi se zrak "naučio"
  - Stanje svijeta će se izmjeniti na način da će NPC postati gospodar zraka, ali više neće ima stvar koju je iskoristio da nauči element *myLocation(airAltar), has(feathers) -> myLocation(airAltar), isMasterOfAir*

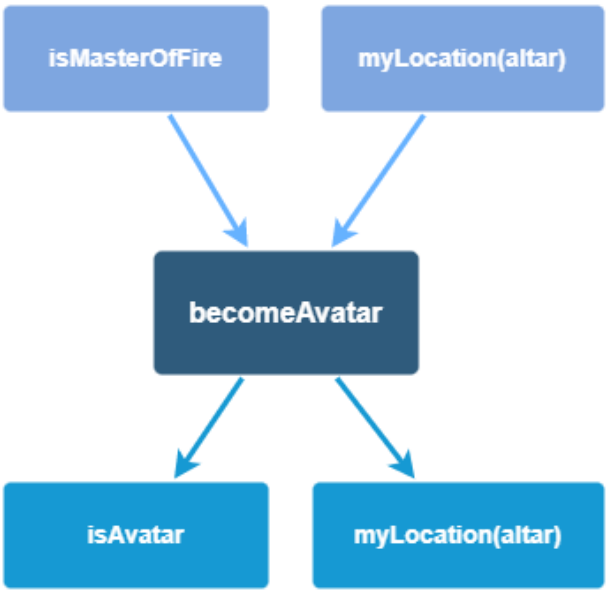
Slika 17: Grafički prikaz funkcije evolveAir (Izvor: autorova izrada, prema [5])

- *evolveWater* je akcija koja služi za učenje elementa vode
  - Kako bi se navedena akcija izvršila sljedeći uvjeti moraju biti ispunjeni: *myLocation(waterAltar), has(wase), isMasterofAir*. Ti uvjeti moraju biti ispunjeni jer se voda može naučiti samo ako je NPC na oltaru vode, ako ima stvar koja je potrebna kako bi se voda "naučila" i ako je "naučio" sve potrebne elementa za "učenje" vode
  - Stanje svijeta će se izmjeniti na način da će NPC postati gospodar vode, ali više neće ima stvar koju je iskoristio da nauči element *myLocation(waterAltar), has(wase), isMasterofAir) -> myLocation(waterAltar), isMasterOfWater*



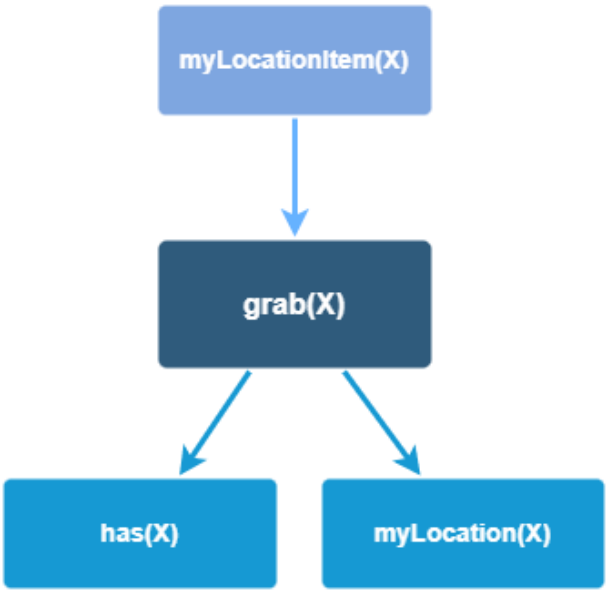
Slika 18: Grafički prikaz funkcije evolveWater (Izvor: autorova izrada, prema [5])

- Akcije *evolveEarth* i *evolveFire* slične su akciji *evolveWater* osim što su za njihovo izvršavanje potrebne druge stvari i oltari
- *becomeAvatar* je akcija koja služi NPC-u da postane Avatar
  - Kako bi se navedena akcija izvršila sljedeći uvjeti moraju biti ispunjeni: *isMasterOfFire, myLocation(altar)*. Ti uvjeti moraju biti ispunjeni jer se Avatarom može postati samo ako su naučeni svi prethodni elementi i ako je NPC na lokaciji posebnog oltara
  - Stanje svijeta će se izmjeniti na način da će NPC postati Avatar*[isMasterOfFire, myLocation(altar)] -> myLocation(altar), isAvatar*



Slika 19: Grafički prikaz funkcije becomeAvatar (Izvor: autorova izrada, prema [5])

- *grab(X)* je akcija koja služi kako bi NPC uzeo stvar X koja mu treba za "učenje" elementa
  - Kako bi se navedena akcija izvršila sljedeći uvjet mora biti ispunjen: *myLocationItem(X)*. Taj uvjet mora biti ispunjen jer NPC mora biti kod stvari kako bi ju mogao uzeti
  - Stanje svijeta će se izmjeniti na način da će NPC posjedovati stvar*[myLocationItem(X) -> has(X), myLocation(X)*



Slika 20: Grafički prikaz funkcije grab (Izvor: autorova izrada, prema [5])

3.1.4.2. Izrada STRIPS algoritma za igru inspiriranu Avatarom

U ovom dijelu opisan je programski kôd algoritma i igre. Također, svi dijelovi programskog kôda opisani su na način da se prvotno predstavlja njihova svrha. Sam programski kôd se može podijeliti na tri dijela:

- STRIPS algoritam napisan u programskom jeziku Prolog
- Umjetna inteligencija za igru i igra u Unreal Engineu
- REST servisi pomoću kojih se povezuje STRIPS algoritam sa Unreal Engineom napisani u programskom jeziku Java

S obzirom da je STRIPS algoritam najvažniji dio rada, opis programskog kôda započet će se s njim. Za početak valja navesti način na koji se sprema trenutno stanje svijeta:

```
myList([myLocation(a)]).
```

U gore navedenom kôdu vidi se lista myList u kojoj se nalazi početno stanje svijeta. Kao što je rečeno u prethodnom poglavlju, prethodno stanje svijeta sastoji se samo od početne lokacije NPC-ja koja može biti bilo koja, a u ovom slučaju se ta lokacija zove *a*. Izvođenjem algoritma lista će se izmjenjivati dok se ne dođe do ciljnog stanja (onog u kojem je NPC Avatar).

Idući dio programskom kôda kojeg valja pokazati su testovi. Testovi služe za pronalazak načina na koji će se stanje dovesti do ciljnog stanja kroz ostala definirana stanja koja predstavljaju među korake, naravno, ukoliko je to moguće. Na prvi se pogled čini kao da je dovoljno imati samo test koji će ispitati na koji način NPC može postati Avatar, međutim, zapravo je potrebno imati puno više testova.

Razlog potrebe za više testova je taj što test određuje akciju koju NPC treba poduzeti i, ukoliko je jedan test pao tj. akcija koja se u slučaju njegova pozitivna izvršavanja može izvršiti sada se ne izvršava, NPC bi radio ništa jer ne postoji test koji bi bio pozitivan i na taj način odredio akciju koju NPC mora izvršiti. Međutim, samo zato što NPC ne može postati Avatar ne znači da ne može napraviti nešto drugo. Primjerice, ukoliko NPC može prikupiti stvari, on bi to trebao i napraviti, bez obzira može li trenutno postati Avatar. Do situacije da NPC ne može postati Avatar, ali može napraviti druge stvari dolazi zbog toga što NPC ne zna u početku gdje se nalaze pojedini predmeti i oltari koji su potrebni kako bi postao Avatar. Kada bi NPC u početku imao te informacije ne bi bilo potrebno imati sve testove već samo spomenuti jedan. Tada bi NPC-ju bilo prelagano izvršiti zadatak.

Testiranje mogućnosti postajanja Avatarom jest jedan od testova koje je moguće izvršiti. Kôd za taj test izgleda ovako:

```
test(Plan):-
    myList(A),
    solve(A, [isAvatar], Plan).
```

Za izvršavanje tog testa potrebno je znati koje je trenutno stanje svijeta. Iz tog razloga se u varijablu A zapisuje koja se stanja nalaze u listi myList. Nakon toga se poziva funkcija solve/3 koja služi za izvođenje STRIPS algoritma. Prvi argument funkcije je lista trenutnih stanja koja je zapisana u varijabli A (iz perspektive izvođenja algoritma to je zapravo lista početnih stanja), drugi argument je lista ciljnih stanja svijeta i treći argument je varijabla u koju se zapisuju akcije koje je potrebno izvršiti kako bi se od početnog stanja došlo do ciljnog stanja. Osim testa mogućnosti postojanja Avatara postoje još testovi koji testiraju mogućnost učenja elementa:

```
testAir(Plan):-
    myList(A),
    solve(A, [isMasterOfAir], Plan).
```

i testovi posjedovanja stvari:

```
testGetFeathers(Plan):-
    myList(A),
    solve(A, [has(feathers)], Plan).
```

Gore nabrojani testovi nisu jedini koji postoje, ali svi ostali testovi mogu pasti u jednu od navedenih kategorija.

Učenjem elemenata i skupljanjem predmeta početno stanje u listi myList se mijenja. Te promijene moraju se nekako zabilježiti u spomenutoj listi. Kako bi se to zabilježilo, potrebno je poslati REST poruku iz Unreal Enginea koju će primiti server napisan u Javi te će taj server pozvati funkciju u Prologu koja će izmijeniti listu na predefiniran način. U nastavku su prikazane i objašnjene funkcije u Prologu koje služe za izmjenjivanje liste.

```
changeLocationToItemLocation(L) :-
        myList(A),
        delete_list([myLocation(_), atItem(L)], A, Remainder),
        append([myLocationItem(L)], Remainder, X),
        list_to_set(X, Set),
        retract(myList(_)),
        asserta(myList(Set)).
```

Iznad ovog odlomka se može vidjeti funkcija changeLocationToItemLocation/1 koja se poziva kada NPC dođe do lokacije na kojoj se nalazi neka stvar. Prva stvar koja funkcija radi jest dohvaćanje liste stanja. Nakon toga funkcija stvara novu listu u kojoj neće biti ona stanja koja više ne vrijede uslijed promijene lokacije. Nova lista sprema se u varijablu Remainder. Potom se stvara još jedna lista u varijabli naziva X kojoj se pridružuju nova stanja. Novostvorena lista u varijabli X može imati više jednakih zapisa. To će se dogoditi u slučaju da NPC dva ili više puta vidi istu stvar ili oltar. Kako bi se rješio taj problem lista X se pretvara u set koji se sprema u varijablu Set. Varijabla Set osigurava da se nikada ne pojavljuje isti zapis. Na kraju se briše sve što se nalazi u listi myList i u nju se zapisuje lista/set koja se nalazi u varijabli Set. Sve ostale funkcije rade na sličan način tako da će u nastavku biti navedene samo funkcije sa

svojim imenom i svrhom bez dodatnog objašnjenja. Funkcija grabbing/1 koja se poziva kada NPC uzme neku stvar:

```
grabbing(L) :-
        myList(A),
        delete_list([myLocationItem(L)], A, Remainder),
        append([myLocation(L), has(L)], Remainder, X),
        list_to_set(X, Set),
        retract(myList(_)),
```

Fukcija changeLocation/1 koja se poziva kada NPC promijeni lokaciju:

```
changeLocation(L) :-
        myList(A),
        delete_list([myLocation(_), at(L)], A, Remainder),
        append([myLocation(L)], Remainder, X),
        list_to_set(X, Set),
        retract(myList(_)),
        asserta(myList(Set)).
```

Funkcija airEvolution koja se poziva kada NPC evoluira:

```
airEvolution :-
        myList(A),
        delete_list([has(feathers), myLocation(airAltar)], A, Remainder),
        append([isMasterOfAir, myLocation(airAltar)], Remainder, X),
        list_to_set(X, Set),
        retract(myList(_)),
        asserta(myList(Set)).
```

Sve ostale funkcije tiču se učenja elemenata i evolucije u Avatara, ali one su slične funkciji airEvolution.

Ne smiju se zaboraviti ni Prolog funkcije koje su ključne za sam STRIPS algoritam. To predstavlja funkcija koja se poziva prilikom testiranja a uključuje i druge funkcije koje se pozivaju tom funkcijom. Prvo se opisuje upravo funkcija solve/3:

```
solve(State, Goal, Plan):-
    solve(State, Goal, [], Plan).

solve(State, Goal, Plan, Plan):-
        is_subset(Goal, State), nl,
        write_sol(Plan).

solve(State, Goal, Sofar, Plan):-
```

```
        opn(Op, Preconditions, Delete, Add, NoMember),
        is_subset(Preconditions, State),
        \+ member(Op, Sofar),
        nonmember(NoMember, State),
        delete_list(Delete, State, Remainder),
        append(Add, Remainder, NewState),
        write(Op), nl,
        solve(NewState, Goal, [Op|Sofar], Plan).
```

Kao što se može vidjeti funkcije solve/3 i solve/4 vrlo su slične određenim funkcijama spomenutim kod problema slaganja blokova (možete usporediti sa funkcijom na str. 13 i 14). Ono što te funkcije rade jest provjeravaju može li se određena akcija izvršiti i, ako može, zapisuju koja je to akcija i mjenjaju trenutno stanje na odgovarajući način. Nakon toga, funkcija poziva samu sebe, ali ovaj put sa ažuriranim stanjem. Ta se funkcija izvršava sve dok se ne pronađu mogući putovi od početnog do ciljnog stanja.

Može se vidjeti da se u funkciji solve/4 spominje funkcija opn/5. Navedena funkcija provjerava može li se određena akcija ispuniti u trenutnom stanju. Prema tome, postoji onoliko opn/5 funkcija koliko je akcija koje NPC može izvršiti. U nastavku se može vidjeti jedna od opn/5 funkcija:

```
opn(goItem(X),
    [myLocation(_), atItem(X)],
    [atItem(X), myLocation(_)],
    [myLocationItem(X)],
        nothing).
```

Iznad navedena opn/5 funkcija služi za provjeru mogućnosti odlaska NPC-a do neke stvari. Prvi argument funkcije govori o kojoj je akciji riječ. Drugi argument funkcije definira stanja koja moraju biti ispunjena kako bi se akcija iz prvog argumenta mogla odraditi. Treći argument navodi stanja koja će biti izbrisana iz liste stanja nakon izvršavanja funkcije dok četvrti argument govori koja će stanja biti upisana u listu nakon izvršavanja akcije. Zadnji peti argument navodi stanja koja ne smiju biti u listi stanja kako bi se ta akcija izvršila. U ovom slučaju ne postoji takvo stanje, stoga u petom argumentu piše nothing, međutim, određene akcije (poput go/1) imaju važeću vrijednost i u petom argumentu:

```
opn(go(X),
    [myLocation(_), at(X)],
    [myLocation(_), at(X)],
    [myLocation(X)],
        atItem(_)).
```

Peti argument funkcije opn/5 služi za upisivanje uvjeta koji ne smiju biti ispunjeni kako bi se izvršila akcija prvog argumenta. Primjerice, gore navedena funkcija ima prvi argument go(X) što znači da funkcija provjerava može li NPC ići na lokaciju na kojoj se ne nalazi stvar

za pokupiti već nešto drugo. Peti argument gore navedene funkcije je atItem(_) što znači da akcija iz prvog argumenta ne može biti poduzeta dok god je NPC svjestan lokacije barem jedne stvari. Osim dvije gore navedene funkcije postoje i funkcije za provjeravanje mogućnosti učenja elementa:

```
opn(evolveAir,
    [myLocation(airAltar), has(feathers)],
    [has(feathers)],
    [isMasterOfAir],
        nothing).
```

Funkcija za provjeravanje mogućnosti postajanja Avatarom:

```
opn(becomeAvatar,
    [isMasterOfFire,myLocation(altar)],
    [isMasterOfFire],
    [isAvatar],
        nothing).
```

Zadnja od opn/5 funkcija koja je prikazana je funkcija za provjeravanje mogućnosti uzimanja neke stvari:

```
opn(grab(X),
    [myLocationItem(X)],
    [myLocationItem(X)],
    [has(X), myLocation(X)],
        nothing).
```

Osim navedenih funkcija s prethodnih stranica za rad STRIPS algoritma potrebne su i funkcije za brisanje elemenata iz liste:

```
delete_list([H|T], List, Final):-
    remove(H, List, Remainder),
    delete_list(T, Remainder, Final).
delete_list([], List, List).

remove(X, [X|T], T).
remove(X, [H|T], [H|R]):-
    remove(X, T, R).
```

Funkcije za stavljanje elemenata u listu:

```
append([H|T], L1, [H|L2]):-
    append(T, L1, L2).
append([], L, L).
```

Funkcije koje provjeravaju jesu li određeni elementi u listi:

```
member(X, [X|_]).
member(X, [_|T]):-
    member(X, T).

nonmember(Arg,[Arg|_]) :-
        !,
        fail.
nonmember(Arg,[_|Tail]) :-
        !,
        nonmember(Arg,Tail).
nonmember(_,[]).
```

Ostale funkcije su ili dovoljno slične spomenutim funkcijama ili nisu toliko bitne za sam STRIPS algoritam pa nisu posebno istaknute.

Sada slijedi kôd u programskom jeziku Java koji služi za stvaranje REST servisa kojima se povezuje Unreal Engine sa Prologom. Prva stvar koja je morala biti rješena u izradi REST servisa jest povezivanje Jave s Prologom. Naime, Java nema ugrađene funkcije koje joj omogućuju povezivanje s Prologom. Iz tog je razloga korištena biblioteka JPL koja povezuje Java virtualni stroj sa SWI-Prologom.[8] Kako bi se navedena biblioteka ugradila u Javu potrebno je postaviti sljedeći kôd u pom.xml od Java projekta:

```
<dependency>
    <groupId>com.github.SWI-Prolog</groupId>
    <artifactId>packages-jpl</artifactId>
    <version>V8.5.7</version>
</dependency>
```

Vrijednost u čvoru *version* može se izmjeniti na najnoviju trenutno dostupnu verziju.

Nakon što je biblioteka s funkcijama za povezivanje s Prologom uključena u projekt može se krenuti sa stvaranjem REST servisa. S obzirom da je najlakše slati JSON objekte REST servisima, stvoren je Plain Old Java Object (u nastavku POJO) koji ima dva String objekta od kojih jedan predstavlja odgovor Prologa, a drugi postavljeno pitanje. Taj POJO će se slati kao odgovor na upite koji se šalju iz Unreal Enginea. Za projekt je napravljena i klasa koja predstavlja REST kontroler koji će primati upite koji se šalju na određene web adrese.

Najbitnija klasa za rad s Prologom jest *PrologConnectionService.java*. U toj klasi se nalaze funkcije kojima se šalje upit iz Unreal Enginea te se taj upit dekodira, saznaje se koji se upit želi postaviti Prologu pa se taj upit i šalje. Kada se primi odgovor iz Prologa, obično se dobiju svi mogući putovi od početnog do ciljnog stanja međutim za rad algoritma potreban je samo najkraći put. Prvo je potrebno vidjeti koji je od putova najkraći pa se onda najkraći put zapisuje u spomenuti POJO iz prethodnog odlomka koji se šalje kao odgovor Unreal Engineu.

Kako bi se uopće mogli slati upiti Prologu potrebno je napisati koji sve upiti postoje. To funkcijonalno ispunjava lista TESTS:

```
private static final List<String> TESTS = List.of("test(Plan).",
    "testFire(Plan).", "testEarth(Plan).",
    "testWater(Plan).", "testAir(Plan).",
    "testGetStaff(Plan)", "testGetWase(Plan)",
    "testGetFireRock(Plan)", "testGetFeathers(Plan)");
```

Kao što se može vidjeti, navedena lista je konstanta. Razlog tomu je što se ona sigurno neće mijenjati pa je najbolja praksa da ona ne bude varijabla već konstanta.

U konstruktoru klase povezuje se sa Prolog datotekom u kojoj je definiran STRIPS algoritam. Povezivanje se odrađuje u konstruktoru zato što je bitno da se Java poveže s navedenom datotekom samo jednom. U nastavku se vidi konstruktor.

```
public PrologConnectionService() {
    String t1 = "consult('D:/sada.pl')";
    Query e1 = new Query(t1);
    e1.hasSolution();
}
```

Prva funkcija koja će se spomenuti jest funkcija:

```
public PrologItem appendToList(String command, String item);
```

Navedena funkcija prima dva argumenta, prvi argument sadrži String u kojem piše koju je akciju izvršio NPC, a u drugi argument pobliže opisuje tu akciju. Primjerice ukoliko u prvom argumentu piše *at*, a u drugom piše *airAltar* to znači da je NPC pronašao oltar zraka i u tom slučaju će se izvršiti sljedeći dio kôda:

```
if(Objects.equals(command, "at")){
    query.setQuery(String.format("place([at(%s)]).", item));
    Query e2 = new Query(query.getQuery());
    e2.allSolutions();
    return query;
}
```

U slučaju ovog programskog isječka varijabla query je prethodno spomenuti POJO, a klasa Query je klasa koja se nalazi u JPL biblioteci i služi za pozivanje Prolog upita. Dakle, ono što se u isječku radi jest zapisuje odgovarajući upit u POJO te se stvara novi upit za Prolog, nakon toga se poziva funkcija koja će poslati novostvoreni upit. U ovom slučaju se radi o zapisivanju stvari u myList listu u prolog pa nije potrebno vratiti odgovor Unreal Engineu.

Ukoliko u prvom argumentu funkcije appendToList piše *retract* to znači da je potrebno izbrisati sve promijene stanja te vratiti listu myList u njezino početno stanje. Programski isječak u kojem se to radi izgleda ovako:

```
if(Objects.equals(command, "retract")){
    query.setQuery(String.format("%s(%s).", command, item));
    Query e2 = new Query(query.getQuery());
    e2.allSolutions();
    query.setQuery("asserta(myList([myLocation(a)])).");
    Query e3 = new Query(query.getQuery());
    e3.allSolutions();
}
```

Kao što se može vidjeti, ovaj programski isječak je sličan prošlom, jedina je razlika što se ovdje šalju dva upita Prologu. Prvi upit služi za brisanje liste myList, a drugi upit služi zavraćanje liste myList u njezino početno stanje. Ostali programski isječci za unošenje stvari u listu su slični prethodnima te kao takvi neće biti posebno prikazani.

Za potrebe testiranja koristila se funkcija

```
public PrologItem getList();
```

koja služi za vraćanje liste stanja. Pomoću te funkcije moglo se vidjeti piše li u listi ono što bi trebalo pisati s obzirom na to što je NPC radio do tog određenog trenutka. Ta funkcija radi na sličan način kao i funkcija *getData* koja je spomenuta u nastavku.

Za kraj je ostala funkcija koja šalje upite koji su zapisani u prethodno spomenutu konstantu. Ta funkcija se poziva kad god se u igri dogodi neka promjena koja utječe na stanja u listi myList. Funkcija izgleda ovako:

```
public PrologItem getData(){
    PrologItem query = new PrologItem();
    for(String test : TESTS){
        query.setQuery(test);
        Query e2 = new Query(query.getQuery());
        Map<String, Term>[] prologResponses = e2.allSolutions();
        Term answer = extractAnswerFromResponse(prologResponses);
        if(answer != null && answer.listLength() > 0){
            query.setAnswer(answer.toString());
            return query;
        }
    }
    return query;
}
```

Funkcija iterira po listi upita koji su spremljeni u konstantu i za svaki element liste šalje odgovarajući upit Prologu. Ukoliko Prolog vrati odgovor u kojem se nalaze upute kako doći do ciljnog stanja, onda se taj odgovor postavlja u POJO koji je za to stvoren te se taj odgovor šalje nazad Unreal Engineu. Iz tog razloga su elementi u listi poredani na način na koji jesu. Naime, u

listu je prvo zapisan upit kojim se provjerava može li NPC postati avatar. Ukoliko NPC zbilja može postati avatar taj će upit odmah vratiti rezultat te se ostali upiti u listi neće izvršavati jer su izvršavanjem upita za postajanje avatarom automatski izvršeni. Ako NPC ne može postati avatar onda će se izvršiti drugi upit u listi koji obuhvaća sve upite osim prvog.

U prethodno navedenoj funkciji spominje se funkcija

```
private Term extractAnswerFromResponse(Map<String,
    Term>[] prologResponses) {
    Term answer = null;
    for (Map<String, Term> response : prologResponses) {
        for (Term term : response.values()) {
            if(answer == null){
                answer = term;
            }
            if(term.listLength() < answer.listLength()){
                answer = term;
            }
        }
    }
    return answer;
}
```

Ta funkcija služi za izvlačenje najkraćeg Prologovog odgovora. Naime, Prolog će pronaći sve moguće načine na koje se može doći od početnog do ciljnog stanja. Ono što je posao ove funkcije jest pregledavanje svih načina i pronalazak onog s najmanje koraka te funkcijsko vraćanje samo tog najkraćeg odgovora.

Naposljetku se prikazuje kôd u Unreal Engineu. U Unreal Engineu je implementiran NPC koji dobija naredbe pomoću STRIPS algoritma. To se radi tako da Unreal Engine šalje zahtjeve preko REST servisa koji se obrađuju i prema tome se šalju upiti Prologu. Nakon toga Prolog vraća rezultate upita koji se vraćaju Unreal Engineu ako je to potrebno, odnosno ako su upiti poslani Prologu promijenili listu stanja.

Unreal Engine koristi REST servise pomoću dodatka zvanog VaRest. Taj dodatak omogućuje korištenje svih mogućih vrsta REST servisa, a olakšava i obradu JSON objekata kako bi se što lakše mogao dobiti potreban podatak. VaRest dodatak preuzet je sa web stranice `https://www.unrealengine.com/marketplace/en-US/product/varest-plugin`.

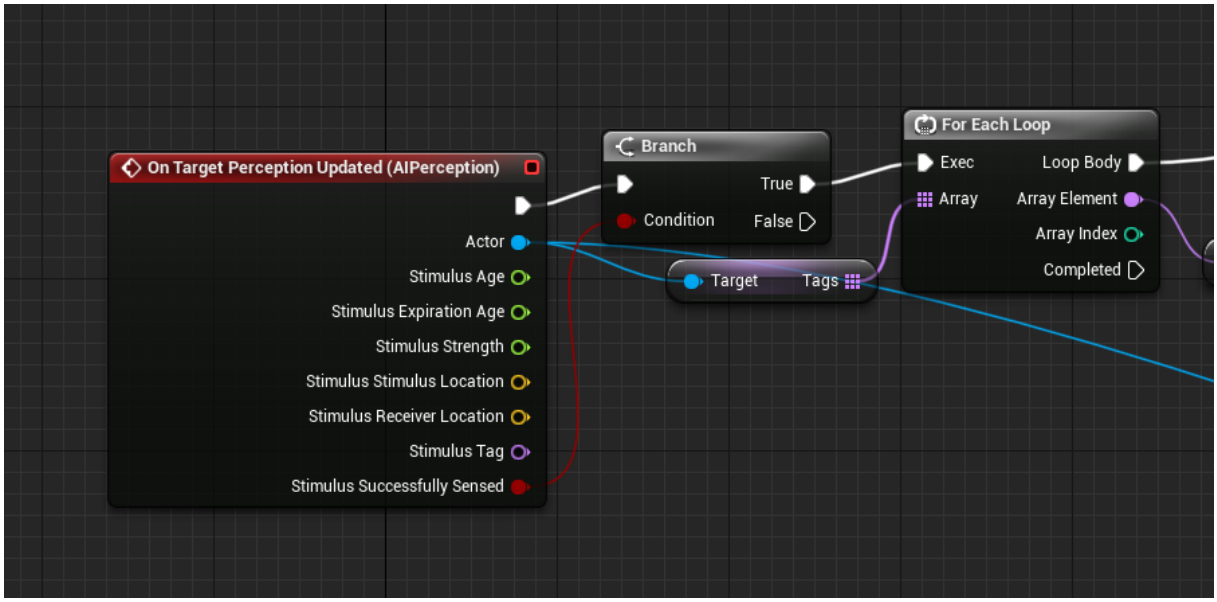
Prvi dio Unreal Engine kôda koji je opisan tiče se AI_Controller klase. Kôd u Unreal Engineu napisan je na poseban način koristeći Unreal Visual Scripting a radi jednostavnosti u nastavku će se korištenje Visual Scriptinga oslovljavati sa Unreal Engine kôdom. Ukratko rečeno, AI_Controller je klasa koja služi kako bi odredila ponašanje nekog *Pawna*. U ovom slučaju, ta se klasa zove AIC_Enemy i služi za pridruživanje stabla ponašanja NPC-ju te za kontroliranje osjetila NPC-ja.

U slučaju ove igre, NPC ima osjetilo vida koje pomoću kojeg može pronalaziti oltare i

stvari koje treba pokupiti. Osjetila se stvaraju pomoću komponente AI_Controllera koja se zove AIPerception. U toj komponenti može se odabrati jedno ili više od ponuđenih osjetila (u ovom slučaju vid) te se mogu postaviti neke dodatne varijable.

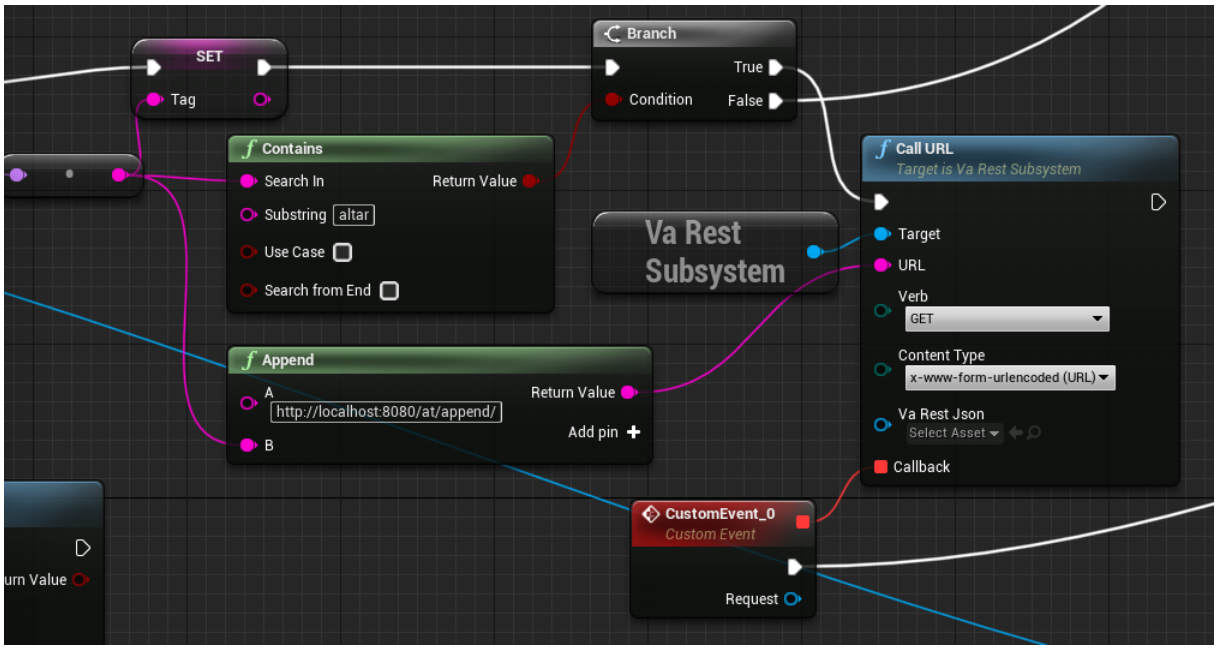
Nakon što je postavljena komponenta AIPerception, NPC može vidjeti stvari. Kako bi se moglo kontrolirati što će se dogoditi kada NPC nešto vidi potrebno je koristiti događaj *On Target Perception Updated(AIPerception)*. Taj događaj se okida svaki put kada NPC nešto vidi ili prestane vidjeti.

Prvo što se radi kada se okine događaj jest provjeravanje je li NPC nešto upravo vidio ili upravo prestao vidjeti. To se radi tako da se provjeri varijabla *Stimulus Successfully Sensed* koja će biti istinita ukoliko je NPC upravo nešto ugledao. Ako je to slučaj, potrebno je vidjeti je li NPC ugledao nešto što je korisno za igru, odnosno je li ugledao neku stvar koju može pokupiti ili neki oltar. Ako je i to istina, onda se šalje odgovarajući REST poziv. REST poziv se šalje preko funkcije *Call URL* koja ima nekoliko bitnih argumenata. Prvi bitan argument je *Target*, u kojeg se u svakom slučaju treba proslijediti referenca na VaRest sustav. Drugi bitan argument je URL koji se u svakom slučaju sastoji od adrese i porta localhosta. Ostatak URL ovisi o tome što je NPC vidio. Ukoliko je NPC vidio oltar nastavak URL bit će */at/append/ime oltara*, a ukoliko je vidio stvar nastavak će biti */atItem/append/ime stvari*. Nastavci na osnovni URL su takvi zato što se u Prologu stvari nalaze u termima pod nazivom *atItem*, a oltari pod termima naziva *at*. Treći bitan argument je *Verb*, u slučaju ove igre, taj argument uvijek poprima vrijednost GET, ali on zapravo treba poprimati vrijednost REST servisa kojeg poziva. Zadnji bitan argument se zove *Callback* a u njemu se nalazi referenca na događaj koji će se pozvati kada REST servis vrati odgovor. Na slici ispod se može vidjeti kako izgleda opisani kôd u Unreal Engineu.

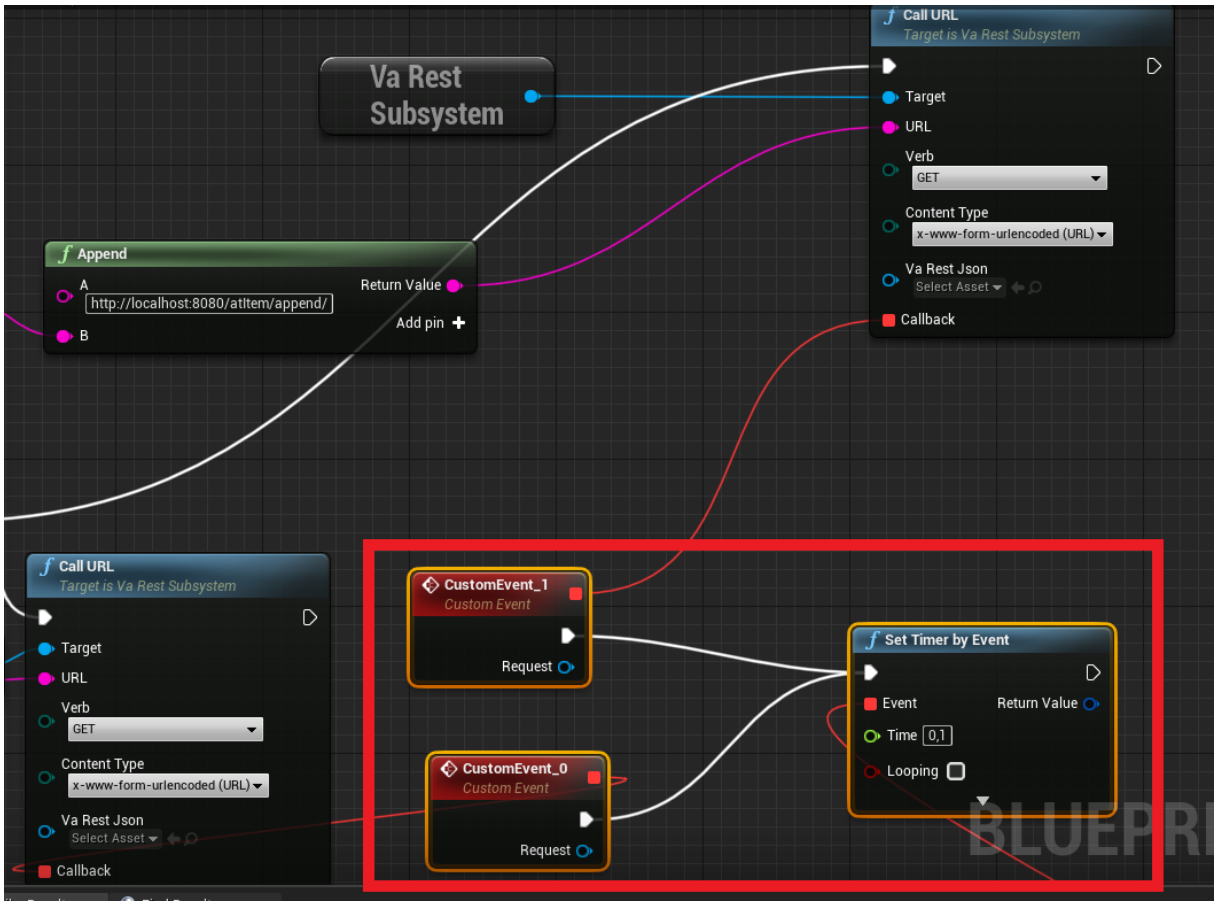


Slika 21: Događaj *On Target Perception Updated(AIPerception)* provjera na početku događaja

Kada se vrati odgovor od REST servisa okida se događaj u kojem se izrađuje timer koji će nakon 100 milisekundi pozvati drugi događaj. To se radi zato da bi se dao određeni odmak između ažuriranja liste u prologu i dohvaćanja novih akcija. Isječak se može vidjeti u slici ispod.



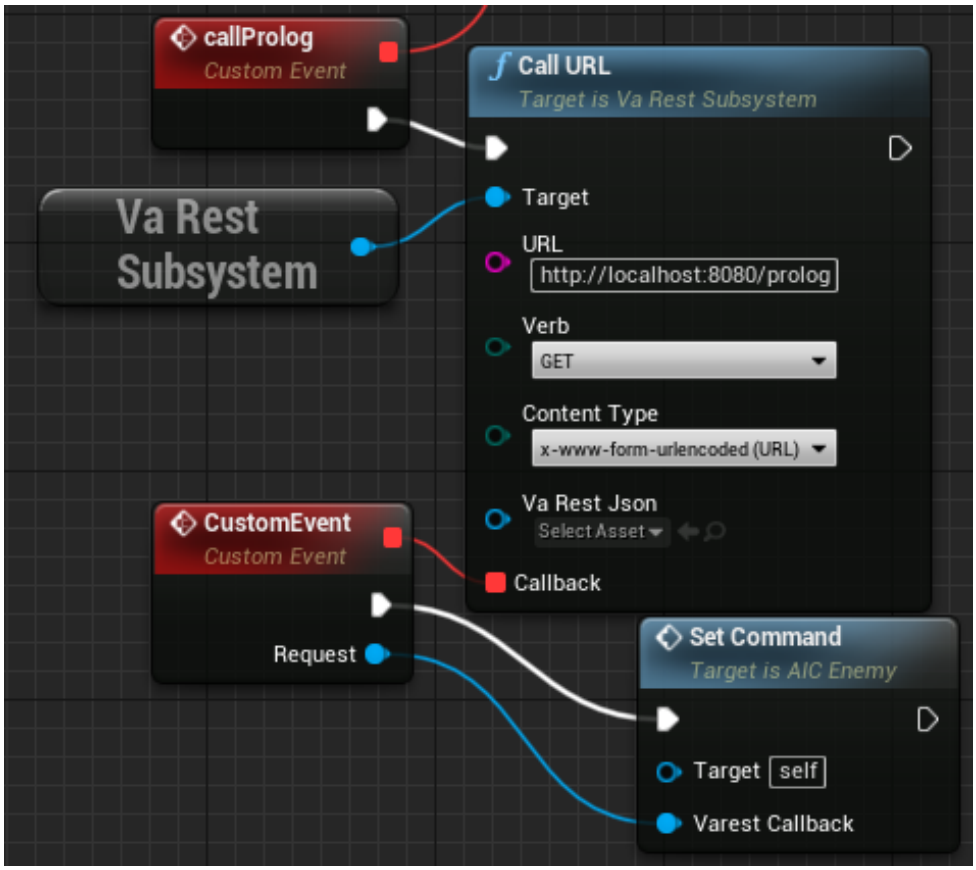
Slika 22: Provjeravanje viđene stvari i pozivanje odgovarajućeg REST servisa (pozivanje REST servisa za stvari izgleda isto)



Slika 23: Postavljanje timera zaokruženo crvenom bojom

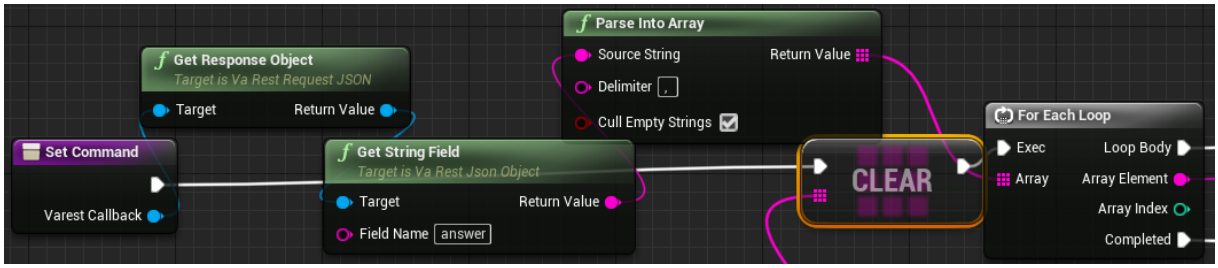
Funkcija koja se poziva timerom služi za stvaranje novog REST poziva. U ovom slučaju radi se o REST pozivu koji će vratiti JSON objekt u kojem će se nalaziti najbolja akcija koju

NPC može poduzeti. URL u drugom argumentu u ovom slučaju ima nastavak */prolog*, a ostatak poziva je isti. Nakon što se vrati odgovor poziva se funkcija *Set Command* u koju se kao argument šalje JSON objekt koji je dobiven kao odgovor. U nastavku se može vidjeti pozivanje REST servisa pomoću funkcije *Call URL* i stvaranje događaja prilikom primanja odgovora. Gorespomenuta funkcija *Set Command* objašnjena je u narednim odlomcima.



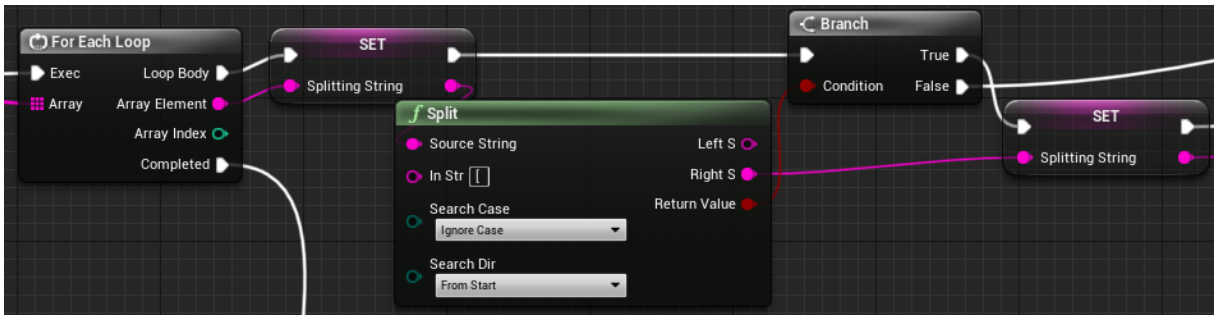
Slika 24: Slanje upita za najbolju akciju

Funkcija Set Command služi za dekodiranje JSON objekta kako bi se saznalo koje akcije kojim redoslijedom NPC treba poduzeti. Prvo se iz JSON objekta uzima polje naziva answer. To je ujedno i polje u koje se u Javi upisuje najkraći put za postizanje ciljnog stanja. Nakon toga se dobiveni String objekt parsa uzimajući zarez kao delimiter. Na taj način se stvara polje u kojem svaki element predstavlja jednu akciju. Postupak opisan u ovom odlomku može se vidjeti na sljedećoj slici.



Slika 25: Parsanje Stringa sa zarezom kao delimiterom

Iako u opisanom trenutku postoji lista sa elementima koji predstavljaju akcije, njihovo uređivanje nije završilo. Naime, neki elementi u sebi imaju znakove koji ne bi trebali biti prisutni u završnoj verziji. Iz tog razloga se iterira po listi te se za svaki element gleda postoje li određeni znakovi. Ako ti znakovi postoje, koristi se funkcija Split kako bi se oni uklonili. Primjer jednog takvog Splita vidi se na slici ispod.

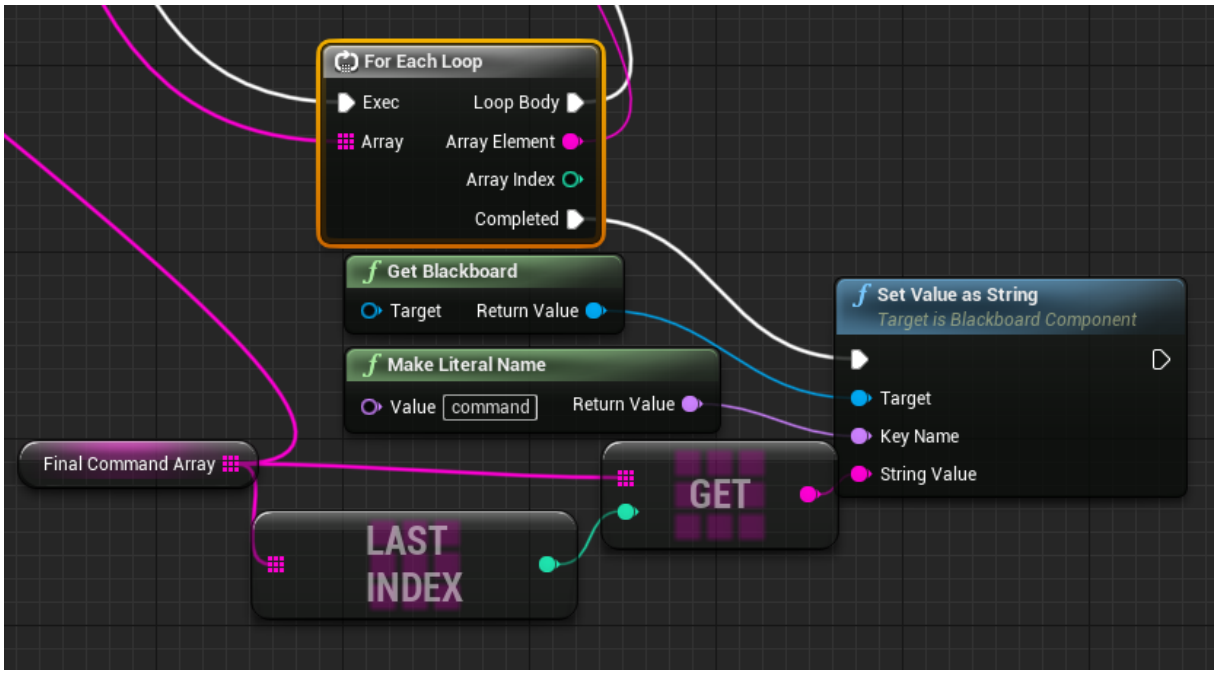


Slika 26: Uklanjanje neželjenih znakova iz elemenata

Nakon što se svaki element uredi, sprema ga se u novu listu. Ta se lista zove Final Command Array i u njoj će biti zapisan svaki element po suprotnom redu od onog po kojem se treba izvoditi. Nakon što se završi iteriranje kroz sve elemente poziva se funkcija koja će postaviti vrijednost Blackboarda na zadnji element u listi Final Command Array. Varijabla Blackboarda u koju će se smjestiti ta vrijednost zove se command i ona služi kako bi stablo ponašanja znalo koju akciju treba izvršiti.

U prethodnom odlomku spomenut je Blackboard. Slikovito rečeno, Blackboard je "mozak" umjetne inteligencije te se u njemu nalazi sve što NPC mora znati.[9] Blackboard je povezan sa stablom odlučivanja koje na taj način može koristiti sve varijable koje su dio Blackboarda. U ovom slučaju Blackboard ima četiri varijable:

- targetLocation -> Vektor u kojem se nalazi lokacija na koju NPC treba doći
- command -> String u kojem se nalazi akcija koju NPC treba izvršiti



Slika 27: Davanje vrijednosti varijabli Blackboarda

- actionNumber -> Integer koji pomaže pri odabiru akcije
- checkAction -> pomoćna varijabla koja daje stablu do znanja kada prestaje sa trenutnom radnjom i ponovo ispituje u koje ce stanje prijeci

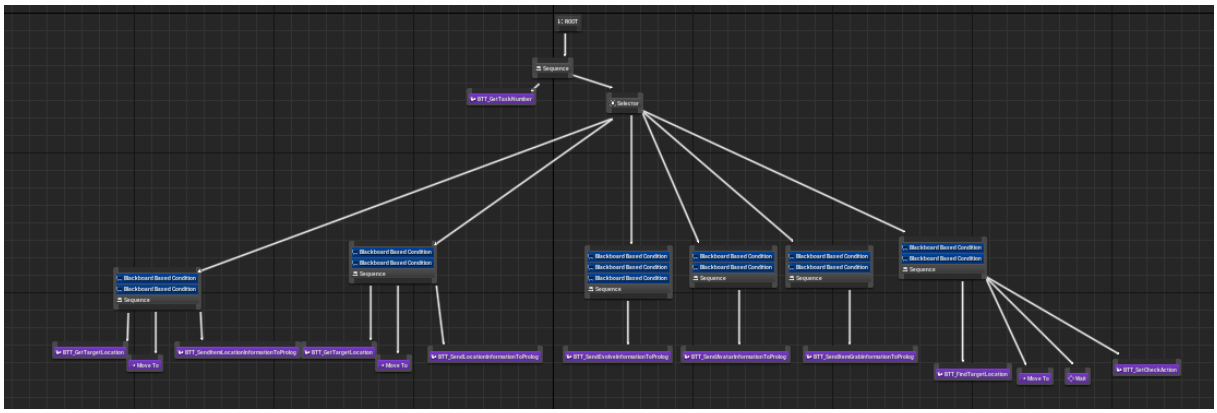
Prilikom objašnjavanja Blackboarda spomenut je još jedan novi pojam - stablo ponašanja. Navedeno stablo služi tomu da umjetna inteligencija može odlučiti na koji čvor treba ići, odnosno koju radnju treba izvršiti.[10] Dakle, radnjama NPC-ja se može direktno upravljati preko stabla.

Na sljedećoj slici se vidi cijelo stablo odlučivanja za napravljenu igru, a kasnije će biti prikazani i objašnjeni pojedini dijelovi. Ono što se može vidjeti na slici jest da se stablo sastoji od korijena koji ima jedno dijete. To dijete ima dva djeteta, prvo u kojem se određuje akcija prema informacijama dobivenim iz Prologa i drugo koje služi za selekciju daljnje akcije. Drugo dijete ima puno djece, a svako od djece predstavlja barem jednu ili više akcija.

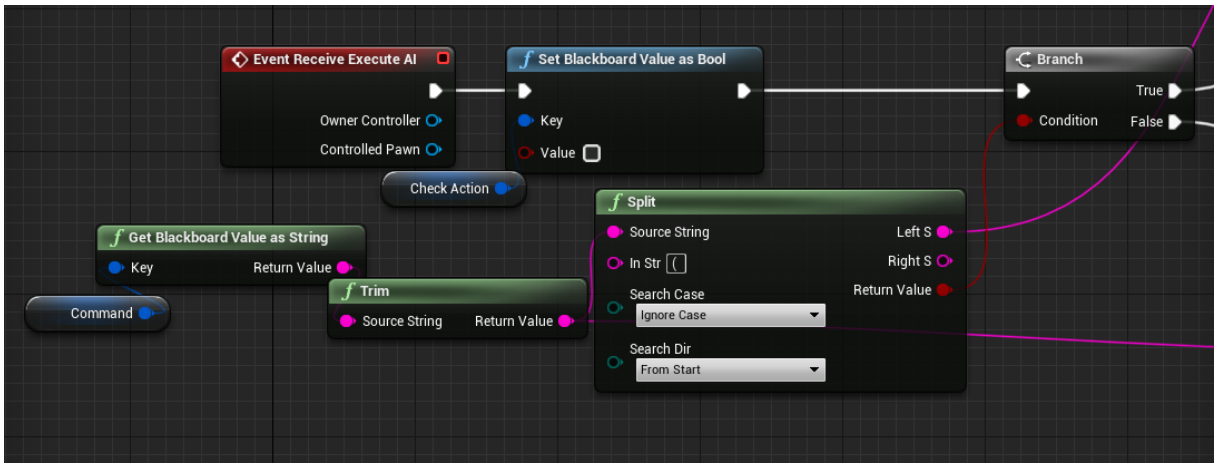
Prva funkcija koja se poziva u stablu je *BTT_GetTaskNumber*. Navedena funkcija služi za određivanje idućeg poteza kojeg NPC treba poduzeti. Funkcija radi na način da uzima vrijednost varijable *Command* Blackboarda te uklanja iz nje bilo kakve suvišne znakove. Prije toga, varijabla *Check Action* Blackboarda postavlja na *False* zato što to označava da se NPC sprema za novi potez.

Uklanjanjem suvišnih znakova vrijednost varijable može se provjeravati u switch metodi. Ovisno o provjerenoj vrijednosti dodjeljuje određeni broj varijabli *Command Action Number* Blackboarda. Ta će se varijabla kasnije koristiti za određivanje poteza kojeg NPC treba poduzeti.

Nakon postavljanja varijable *Command Action Number* postoji još jedna stvar koja se

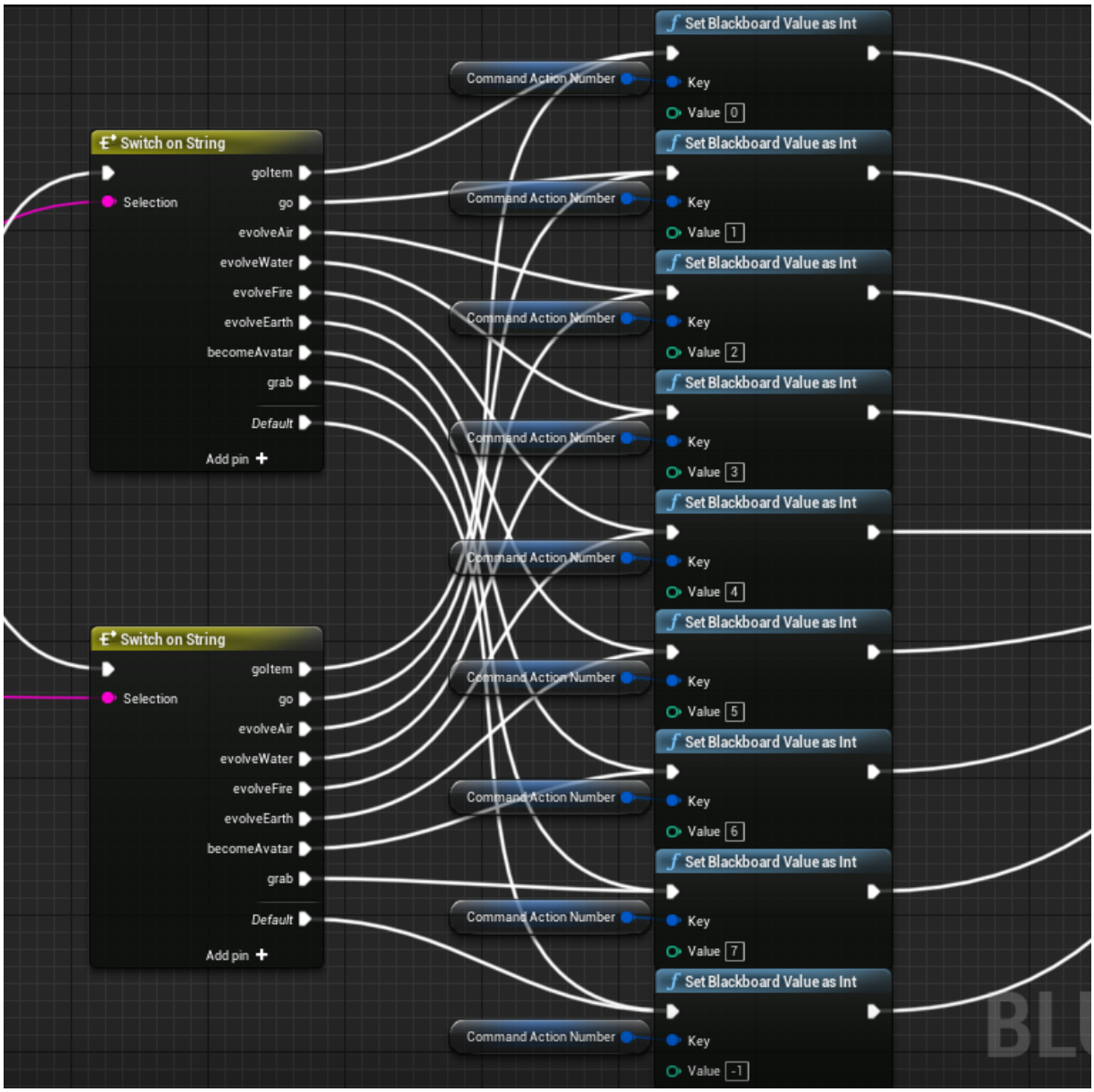


Slika 28: Stablo odlučivanja



Slika 29: Uklanjanje znakova i postavljanje *Check Action*

mora napraviti u funkciji. Naime, bitno je da u listi u Prologu svaka od stvari i oltara bude zapisana samo jednom. Kako bi se to osiguralo, potrebno je onemogućiti pokretanje događaja u kojem se poziva REST servis za zapisivanje novih stvari u listu. Taj se događaj poziva kada NPC ugleda neku stvar ili oltar pa je stoga potrebno makniti stvari i oltare iz percepcije NPC-ja. To se radi tako da se dohvati referenca na pojedinog actora te se pomoću funkcije *Unregister from Perception System* taj actor uklanja iz percepcije. Navedeni kôd se može vidjeti na idućoj slici.

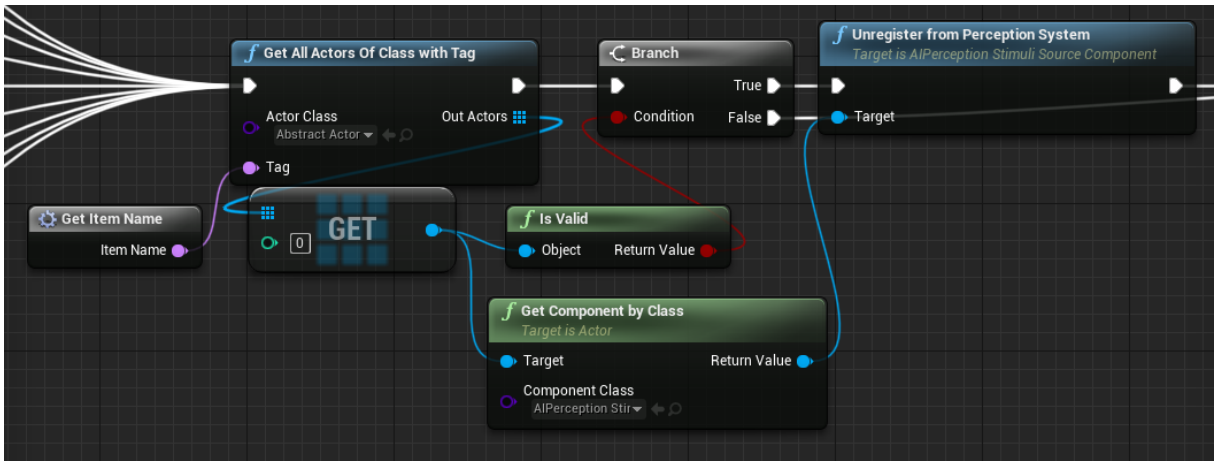


Slika 30: Switch i dodijeljivanje vrijednosti *Command Action Number*

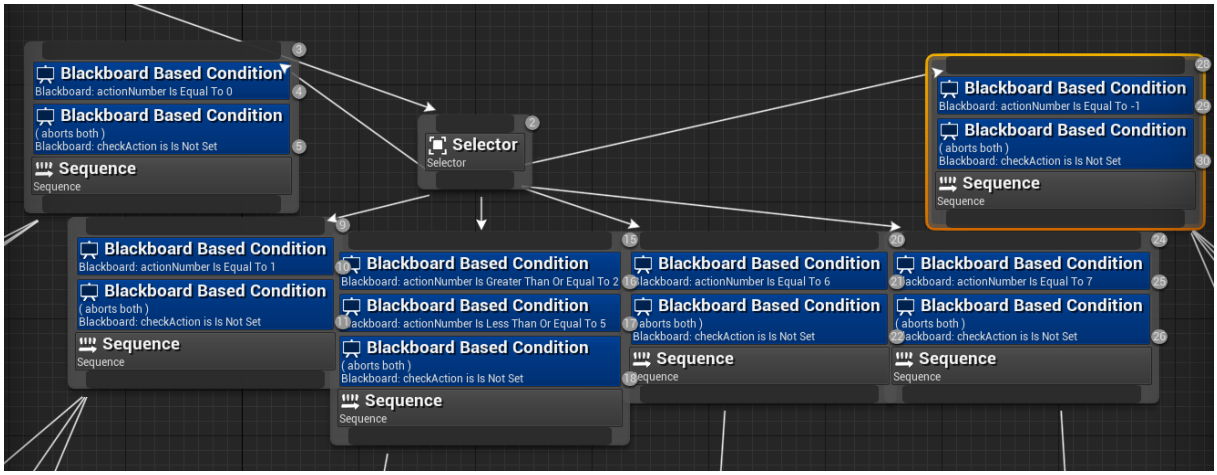
Nakon što se, izvršavanjem funkcije *BTT_GetTaskNumber*, odredi vrijednost varijable *Command Action Number* stablo može navigirati do čvorova koji se trebaju izvršiti. To se događa na način da se nakon zadatka *BTT_GetTaskNumber* nalazi čvor selektor koji ima šestero djece. Svako od djece je čvor sekvence sa dekoratorima. Izvršava se prvo dijete kojem svi dekoratori ispunjavaju uvjete.

Prva sekvenca će se izvršiti ukoliko je *actionNumber* jednak nuli i ukoliko varijabla *checkAction* nije postavljena. To će se dogoditi ukoliko NPC treba ići na lokaciju određene stvari. Kod izvršavanja sekvence pokrenut će se tri funkcije: *BTT_GetTargetLocation*, *Move To* i *BTT_SendItemLocationInformationToProlog*. Za razliku od ostale dvije, funkcija *Move To* je ugrađena u Unreal Engine i služi kako bi character pomicao sa trenutnog na argumentom definirano mjesto.

Funkcija *BTT_GetTargetLocation* služi za postavljanje lokacije na koju će se NPC po-

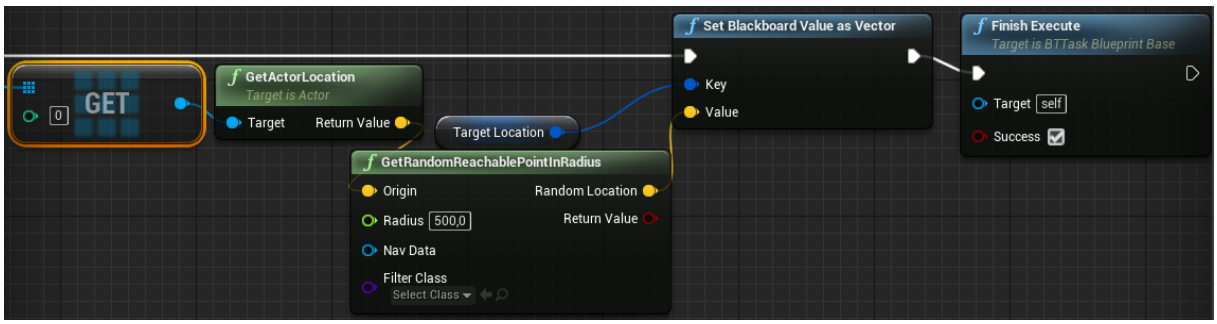


Slika 31: Uklanjanje actora iz percepcije



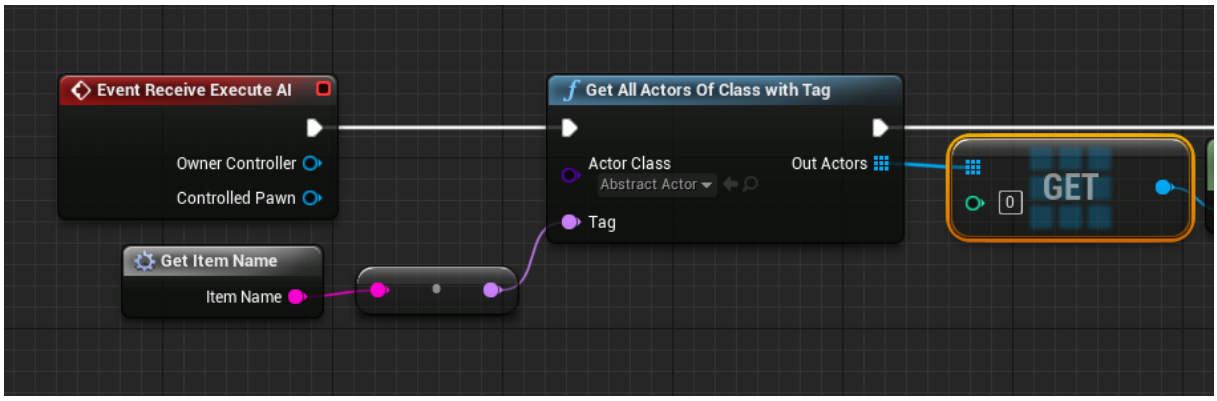
Slika 32: Dekoratori koji moraju biti ispunjeni da bi se izvršila djeca pripadne sekvence

maknuti. Početno, u funkciji se dohvaća tag actora do kojeg NPC treba doći. Nakon toga se uzima referenca na actora pomoću tag-a. Koristeći referencu može se dobiti lokacija actora koja se upisuje u varijablu Blackboarda *Target Location*.



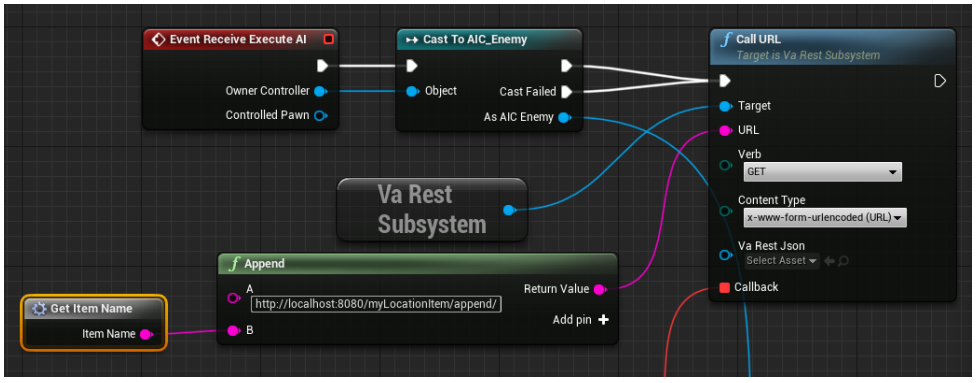
Slika 33: Davanje vrijednosti varijabli Target Location

Ovdje je važno napomenuti da funkcija *Get All Actors Of Class with Tag* funkcionira na način da je potrebno zadati klasu actora po kojoj se pretražuje i tag actora kojeg se želi dobiti. Sve stvari i oltari do kojih se može doći imaju zajedničku klasu actor, međutim svi ostali objekti u igri također imaju tu klasu. Iz tog razloga su svi oni objekti koji su bitni za gameplay naslijedili vlastito izrađenu klasu Abstract Actor koju nasljeđuju samo oni. Na taj način funkcija *Get All Actors Of Class with Tag* pretražuje desetak actora, a ne sve pa su performanse značajno bolje.



Slika 34: Korištenje klase Abstract Actor umjesto općenitije klase Actor

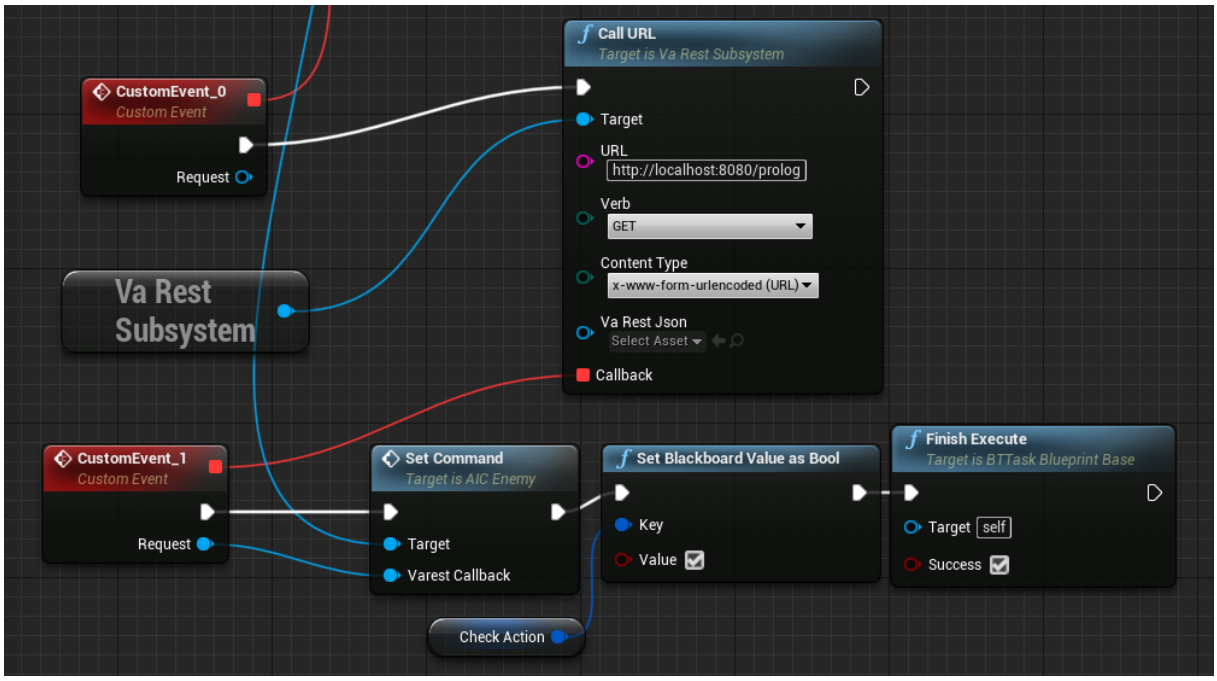
Funkcija *BTT_SendItemLocationInformationToProlog* služi za slanje REST upita kako bi se unijele promijene i dobile nove informacije. Prvi REST upit se šalje na url s nastavkom */myLocationItem/append/Ime stvari*. Pomoću navedenog nastavka će se znati da je listu potrebno promijeniti tako da se doda novi *myLocationItem* određenog naziva.



Slika 35: Slanje REST upita

Pomoću drugog REST upita koji se šalje funkcijom *BTT_SendItemLocationInformationToProlog* dobija se popis akcija koje je potrebno izvršiti s obzirom na novu listu stanja. Nakon toga poziva se funkcija *Set Command* u AI_Controlleru koja je spomenuta ranije. Za kraj, postavlja se varijabla Blackboarda *Check Action* na True što označava da je radnja završena i da se sekvenca prekida.

Druga sekvenca će se izvršiti ukoliko je *actionNumber* jednak jedan i ukoliko varijabla *checkAction* nije postavljena. To će se dogoditi ukoliko NPC treba ići na lokaciju oltara. Kod izvršavanja sekvence pokrenut će se tri funkcije: *BTT_GetTargetLocation*, *Move To* i *BTT_SendLocationInformationToProlog*. Funkcija *BTT_SendLocationInformationToProlog* je vrlo slična funkciji *BTT_SendItemLocationInformationToProlog* pa nije potrebno ponovno



Slika 36: Dobivanje nove liste akcija

objašnjavati na koji način radi.

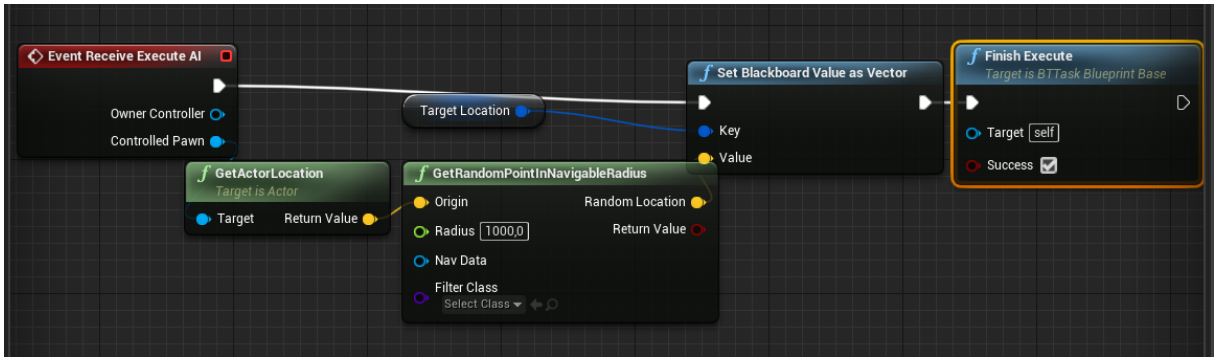
Treća sekvenca će se izvršiti ukoliko je *actionNumber* veća od dva ili manja od pet i ukoliko varijabla *checkAction* nije postavljena. To će se dogoditi ukoliko NPC treba naučiti neki od elemenata. Kod izvršavanja sekvence pokrenut će se funkcija *BTT_SendEvolveInformationToProlog*. Funkcija *BTT_SendEvolveInformationToProlog* je vrlo slična funkciji *BTT_SendItemLocationInformationToProlog* pa nije potrebno ponovno objašnjavati na koji način radi.

Četvrta sekvenca će se izvršiti ukoliko je *actionNumber* jednak šest i ukoliko varijabla *checkAction* nije postavljena. To će se dogoditi ukoliko NPC treba evoluirati u Avatara. Kod izvršavanja sekvence pokrenut će se funkcija *BTT_SendAvatarInformationToProlog*. Funkcija *BTT_SendAvatarInformationToProlog* je vrlo slična funkciji *BTT_SendItemLocationInformationToProlog* pa nije potrebno ponovno objašnjavati na koji način radi.

Peta sekvenca će se izvršiti ukoliko je *actionNumber* jednak sedam i ukoliko varijabla *checkAction* nije postavljena. To će se dogoditi ukoliko NPC treba uzeti neku stvar. Kod izvršavanja sekvence pokrenut će se funkcija *BTT_SendItemGrabInformationToProlog*. Funkcija *BTT_SendItemGrabInformationToProlog* je vrlo slična funkciji *BTT_SendItemLocationInformationToProlog* pa nije potrebno ponovno objašnjavati na koji način radi.

Šesta sekvenca će se izvršiti ukoliko je *actionNumber* jednak -1 i ukoliko varijabla *checkAction* nije postavljena. To će se dogoditi ukoliko NPC nije dobio nikakvu instrukciju od algoritma. Kod izvršavanja sekvence pokrenut će se četiri funkcije: *BTT_FindTargetLocation*, *Move To*, *Wait* i *BTT_SetCheckAction*.

Funkcija *BTT_FindTargetLocation* služi za pronalaženje neke lokacije u određenom radijusu oko NPC-ja. Ta se funkcija poziva ukoliko NPC ne može raditi ništa drugo. Navedena funkcija radi na način da pomoću reference na NPC-ja dohvaća njegovu lokaciju. Vektor pronađene lokacije se mjenja pomoću funkcije *GetRandomPointInNavigableRadius* koja daje neku drugu lokaciju u određenom radijusu. Završno, varijabla Blackboarda Target Location se postavlja na nasumično dobivenu lokaciju.



Slika 37: Funkcija *BTT_FindTargetLocation*

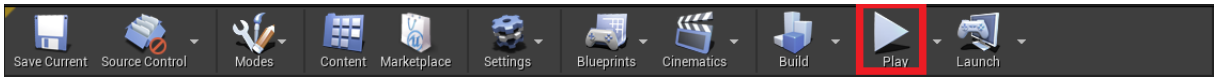
Zadnja bitna funkcija je *BTT_SetCheckAction* koja služi za postavljanje vrijednosti varijable *Check Action*. Funkcija je vrlo kratka te je u njoj samo jedna metoda naziva *Set Blackboard Value as Bool*. U prethodno navedenoj funkciji postavlja se varijabla *Check Action* na *True* i to je sve što se u njoj radi.

3.1.4.3. Primjer pokretanja igre

Za pokretanje igre potrebno je imati uključen Unreal Engine i Intellij IDEA. Pritom je potrebno prvo uključiti server na kojem se nalaze REST servisi (Intellij IDEA) zato što igra koja se pokreće u Unreal Engineu bez toga neće ispravno raditi. Spomenuti server može se jednostavno pokrenuti klikom na play ikonu koja se nalazi u gornjem desnom kutu Intellija te se nalazi na slici ispod na kojoj je odgovarajuća ikona zaokružena crvenom bojom.

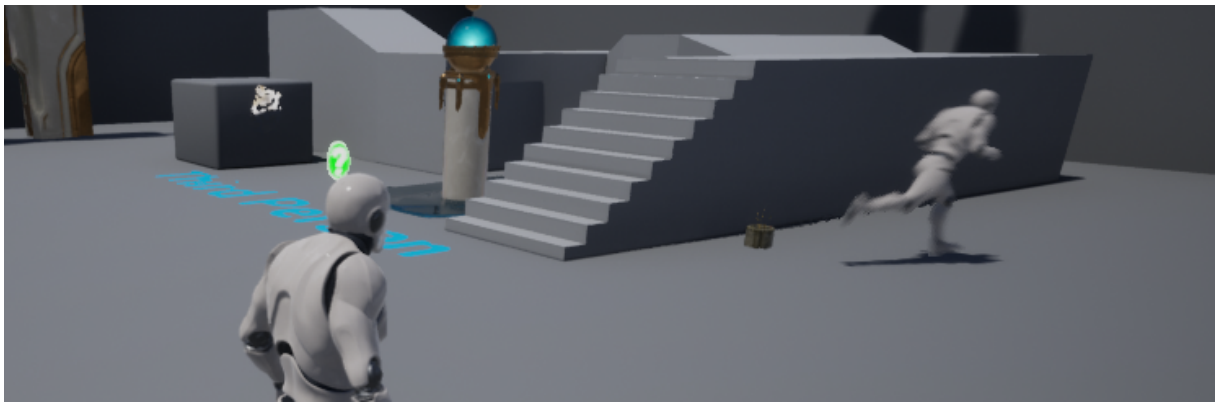
Slika 38: Pokretanje servera pomoću Intellija

Nakon što je server pokrenut, može se pokrenuti i igra. Igra se pokreće tako da se ode na njezin level editor te se klikne na play ikonu. Level editor je ujedno i prva stvar koja se prikaže kada se pokrene projekt s igrom. Play ikona za pokretanje igre nalazi se na alatnoj traci u gornjem dijelu ekrana iznad viewporta. Može se vidjeti na sljedećoj slici zaokružena crvenom bojom.



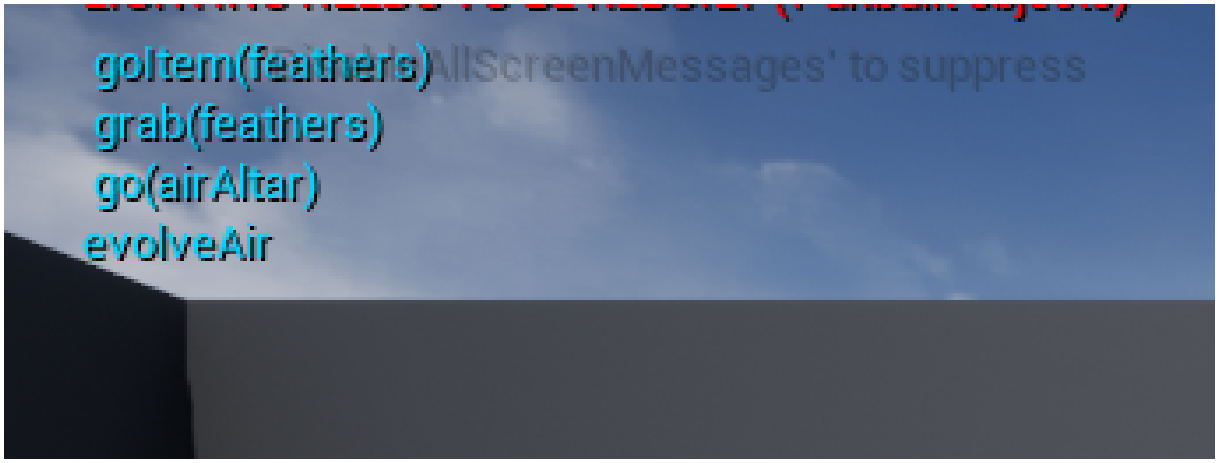
Slika 39: Pokretanje igre pomoću Unreal Enginea

Pošto je prikazano na koji način je potrebno pokrenuti igru, može se prijeći na prikaz same igre. Prva stvar koja će biti prikazana jest traženje. Naime, dok NPC nije u mogućnosti učenja elementa niti je našao neku stvar, onda ide na nasumične lokacije kako bi pokušao naći neku od stvari ili oltare. To se može vidjeti na sljedećoj slici.



Slika 40: NPC traži stvari i oltare

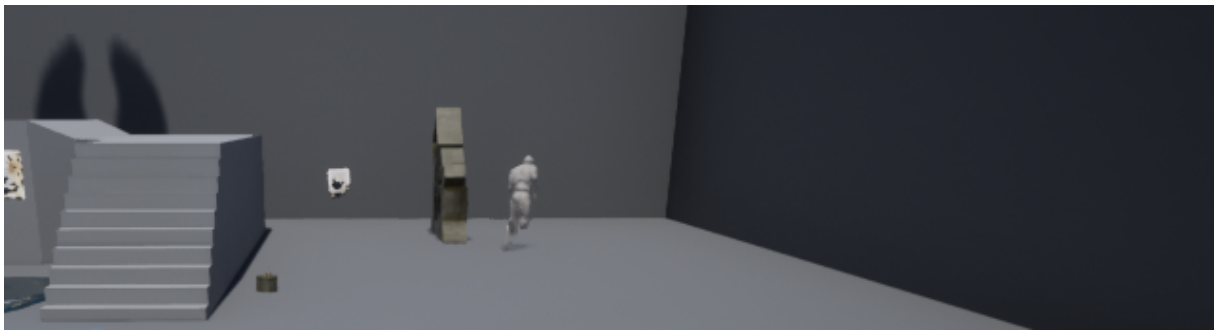
Nakon što je NPC našao stvari ili oltare, REST servis će vratiti poruku u kojoj će biti ispisane akcije koje NPC treba izvršiti. Sam REST servis dobit će te podatke od STRIPS algoritma koji je implementiran u Prologu. Način na koji to funkcionira objašnjen je u prethodnim poglavljima. Upute koje Unreal Engine prima vidljive su na sljedećoj slici.



Slika 41: Akcije koje je potrebno izvršiti

Nakon što NPC primi akcije koje treba izvršiti on će krenuti u skupljanje stvari i učenje elemenata. Na sljedećoj slici se može vidjeti kako NPC ide u smjeru oltara zemlje kako bi naučio taj element. NPC na isti način ide i prema oltarima ostalih elemenata kada su uvjeti za to ispunjeni.

Za kraj, nakon što je NPC naučio sve elemente i zna gdje se nalazi poseban oltar, on može ići prema njemu evoluirati. Igra je implementirana na način da se na ekranu ispiše poruka AVATAR!!!! nakon što NPC dođe do posebnog oltara i postane Avatarom. Na idućoj slici se može vidjeti trenutak nakon što je NPC postao Avatarom.



Slika 42: Odrađivanje dobivenih akcija

Slika 43: NPC postaje Avatar

4. Zaključak

Izrađeni algoritam dobro je odradio svoj posao. Podaci su dohvaćeni brzo i nije bilo logičkih grešaka kod korištenja vanjskih podataka za potpuno ili djelomično upravljanje kojih zna biti kada se umjetna inteligencija izrađuje u samom Unreal Engineu. Za nekoga tko radi s Prologom izrada algoritma za ovakvu svrhu ne bi trebala biti problem zato što nema mnogo koncepata koje je potrebno znati kako bi se algoritam izradio. Pozitivna stvar je i da se dobar dio algoritma može ponovno iskoristiti kada bi se izradio algoritam za neku drugu svrhu. Također, izrađeni algoritam teoretski bi se mogao koristiti za razne vrste igara.

Jedno od svojstava algoritma je neefikasnost. Funkcije za provjeravanje akcija morale su biti izrazito optimizirane jer bi inače sam algoritam tražio rješenje više sekundi. Također, s većim brojem akcija i stanja algoritam postaje sve sporiji. Stoga se postavlja pitanje, koliko bi algoritam bio učinkovit kada bi se koristio za izradu neke veće igre.

Izrada algoritma poput STRIPS algoritma vrlo je pozitivna za učenje programskog jezika Prolog jer se na internetu nalazi dovoljan broj primjera jednostavnih STRIPS algoritama koji pojedinca mogu uvesti u temu. Kako bi se izradio vlastiti algoritam potrebno je znati na koji način rade pojedini konstrukti u Prologu, kako se izvršavaju predikati te se potrebno upoznati s nekim od ugrađenih predikata Prologa. Iako izrada ovakvog algoritma pruža dobru priliku za učenje, bilo bi teško koristiti algoritam za izradu igre zato što je Prolog relativno nepoznat programski jezik kojeg ne zna puno ljudi. Iz tog razloga je nepraktično da velika poduzeća koja proizvode igre koriste navedeni algoritam jer bi morali potrošiti puno vremena kako bi našli ili educirali zaposlenike o programiranju u Prologu ili o STRIPS algoritmu. Naravno, ukoliko se radi o malom indie projektu, moguće je koristiti STRIPS algoritam ali je potrebno paziti da bude dovoljno optimiziran kako *gameplay* ne bi patio zato što NPC-ju treba previše vremena da napravi radnju .

Još jedna pozitivna strana ovog projekta je i činjenica da je potrebno raditi u većem broju alata i biblioteka što je korisno za učenje korištenih alata. Osim rada u Prologu, za ostvarivanje ovog projekta bilo je potrebno raditi i u Unreal Engineu i Javi. Prema tome, potrebno je naučiti na koji način radi umjetna inteligencija u Unreal Engineu, kako rade REST servisi u Unreal Engineu i Javi, kako se u Javi programiraju vršne točke itd. To može služiti kao pomoć u budućem obrazovanju i poslu jer se stječe znanje u mnogim područjima, a može se vidjeti i što pojedincu bolje *leži* i što mu je zanimljivije.

Za kraj, može se reći da je izrada ovog projekta bila korisna jer je omogućila upoznavanje s mnogobrojnim tehnologijama te je pružila uvid za izradu igara. Ovaj je projekt posebno zanimljiv za one koji se zanimaju u izradu igri ili one koji žele naučiti nešto o drugačijem programskom jeziku od onih koji se obično koriste. Također, teorijski dio ovog projekta relativno je mali u odnosu na praktični dio što ga čini idealnim za sve one koji više vole praktične radove od teorijskih.

Popis literature

[1] R. E. Fikes i N. J. Nilsson, „STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," 1971.

[2] K. Becker, *Artificial Intelligence Planning with STRIPS, A Gentle Introduction*, 2015. adresa: http://www.primaryobjects.com/2015/11/06/artificial-intelligence-planning-with-strips-a-gentle-introduction/ (pogledano 16. 6. 2022.).

[3] B. Logan, „AI Programming Techniques Lecture 13 : Planning," *School of Computer Science*, str. 30, 2012. adresa: http://www.cs.nott.ac.uk/$%5Csim$pszbsl/G52APT/slides/13-Planning.pdf.

[4] M. Schatten, „Višeagentni sustavi Agenti sa sposobnošć praktičnog rezoniranja," *ELF*, 2021.

[5] N. J. (J. Nilsson, *Artificial intelligence / by Nils J. Nilsson.* Serija Technical note (SRI) ; 89. Menlo Park, Calif: Artificial Intelligence Center, SRI International, 1974.

[6] D. van Berkel, *Using STRIPS to make plans in block world*, 2020. adresa: https://swish.swi-prolog.org/p/STRIPS%20Block%20World.swinb.

[7] *Actions and Change-p. 1/45*. adresa: https://www.cs.ru.nl/$%5Csim$peterl/teaching/KeR/actions.pdf (pogledano 22. 6. 2022.).

[8] *JPL - Introduction*. adresa: https://jpl7.org/ (pogledano 26. 6. 2022.).

[9] *Behavior Tree Quick Start Guide | Unreal Engine Documentation*, 2022. adresa: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/ArtificialIntelligBehaviorTrees/BehaviorTreeQuickStart/ (pogledano 27. 6. 2022.).

[10] *Behavior Trees | Unreal Engine Documentation*, 2022. adresa: https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/ (pogledano 27. 6. 2022.).

Popis slika