Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Simon Mihevc

Večagentno iskanje poti v strateških igrah

diplomsko delo na univerzitetnem študiju
univerzitetni program računalništvo in informatika

Mentor: akad. prof. dr. Ivan Bratko

Ljubljana, 2014

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Tipičen problem pri strateških igrah je iskanje poti med dvema točkama v prostoru igre. V primeru, da se po poti premika en agent, lahko za iskanje poti uporabimo standardni algoritem A*. Če pa se po poti premika več agentov, ali več enot, govorimo o večagentnem iskanju poti. Za večagentno iskanje je potrebno osnovni algoritem nadgraditi. V diplomskem delu raziščite take prilagoditve v literaturi in implementirajte nekaj izbranih algoritmov. Uspešnost teh algoritmov eksperimentalno ovrednotite in po možnosti obstoječe algoritme izboljšajte.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Simon Mihevc, z vpisno številko **63040102**, sem avtor diplomskega dela z naslovom:

Večagentno iskanje poti v strateških igrah

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. Ivana Bratka,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 11. novembra 2014 Podpis avtorja:

Zahvaljujem se mami in stricu za materialno in moralno podporo, mentorju profesorju Ivanu Bratku za pomoč pri pisanju, kolegom in kolegicam s faksa za spodbude pri pisanju, ožjemu sorodstvu, ki me je pripeljalo na pot računalništva, tako da so mi skupaj kupili prvi računalnik in na koncu še Anteji za podporo in potrpljenje.

Kazalo

Slike

Tabele

Povzetek

Iskanje poti je klasično področje umetne inteligence, uporabno za reševanje problemov v robotiki, vojaški industriji, računalniških igrah itd. Znani hevristični algoritmi, kot je A*, dobro rešujejo enoagentno iskanje poti. Večagentno iskanju poti z algoritmom A* rezultira v kombinatorični eksploziji, posledica česar je neuporabnost algoritma že pri majhni skupini agentov. Višjenivojski algoritmi v kombinaciji z algoritmom A* problem iskanja rešujejo bistveno bolje.

V tej diplomski nalogi sem se ukvarjal z izdelavo, primerjavo in izboljšanjem algoritmov za večagentno iskanje poti na domeni, ki je tipična za realno časovne strateške igre. Izbral sem dva znana algoritma - večagentno iskanje poti z uporabo hodnika in večagentno iskanje poti z detekcijo neodvisnosti in dekompozicijo operatorjev - in jih implementiral po opisu v raziskovalnih člankih. Algoritma sem primerjal glede na metrike kot so čas računanja poti in povprečne razdalja med agenti. Oba algoritma sta imela določene pomankljivosti. Algoritem Večagentno iskanje poti z uporabo hodnika ni ohranjal enotnosti skupine agentov. Algoritem Večagentno iskanje z detekcijo neodvisnosti in dekompozicijo operatorjev je potreboval preveč časa za iskanje rešitve. Pomanjkljivosti obeh algoritmov sem v precejšnji meri odpravil. Rezultati implicirajo, da sta izboljšana algoritma, uporabna v realno časovnih strateških igrah.

Ključne besede: Večagentno iskanje poti, realno časovne strateške igre, algoritem A*, večagentno iskanje poti z uporabo hodnika, večagentno iskanje poti z detekcijo neodvisnosti in dekompozicijo operatorjev.

Abstract

Pathfinding is a classical field of artificial intelligence, used for solving problems in robotics, military industry, computer games etc. Known heuristic algorithms, such as A*, solve the problem of single agent pathfinding. Multiagent pathfinding using A* algorithm results in combinatorical explosion, which makes A* algorithm in its basic form useless for anything but small agent groups. Using higher level algorithms that incorporate A* algorithm makes search more efficient.

In this thesis I worked on creating, comparing and improving algorithms for multi-agent path planning on a domain typical for real-time strategy games. I chose two known algorithms - Multiagent pathfinding using clearance and Multiagent pathfinding using independence detection and operator decomposition - and implemented them according to their descriptions in research papers. I compared the algorithms using metrics such as pathfinding time and average distance between agents. Both algorithms had certain problems. Multiagent pathfinding using clearance had issues with maintaining group compactness. On the other hand multi-agent pathfinding using independence detection and operator decomposition had issues with high time complexity. I considerably improved efficiency of both algorithms. The results imply, that the improved algorithms are useful in real-time strategy games.

Keywords: multi-agent pathfinding, real-time strategy gamess, A* algorithm, multi-agent pathfinding using clearance, multi-agent pathfinding using independence detection and operator decomposition.

Poglavje 1

Uvod

1.1 Večagentno iskanje poti

Večagentno iskanje poti ali VaIP (ang. Multi Agent Pathfindind ali MAPF) je oblika planiranja, katere cilj je poiskati nekonfliktne poti za skupino agentov [9]. Uporablja se ga v robotiki, usmerjanju prometa, računalniških igrah itn. Formalno se ga definira na sledeč način. Naj bo podan graf $G =< V, E >$, kjer V označuje vozlišča ter E povezave med njimi, in k agentov z znanimi začetnimi položaji in podanimi ciljem. Problem je poiskati pot za vsakega agenta, ki ga pripelje od začetnega položaja v bližino cilja. Problem je v splošnem NP-poln [8], ker je posplošitev problema drseče sestavljanke (ang. sliding puzzle). Posledica algoritmične težavnosti je, da algoritmi tipično iščejo suboptimalne rešitve. Cilj diplomske naloge je implementacija, primerjanje in izboljšava algoritmov, ki rešujejo problem večagentnega iskanja poti na domeni strateških iger.

Izbral sem dva algoritma, ki ilustrirata dva različna pristopa k reševanju tega problema. Prvi algoritem - Večagentno iskanje poti z uporabo hodnika - poišče zgolj eno glavno pot in uporablja enostavna pravila za usmerjanje agentov vzdolž poti. Predor ponazarja področje okoli glavne poti, znotraj katerega se agenti lahko gibajo. Algoritem agente tretira kot sistem delcev s silami, položaji in hitrostmi, ki ga rešuje z metodo Runge-Kutta.

Drugi algoritem - Večagentno iskanje poti z detekcijo neodvisnosti in dekompozicijo operatorjev - poišče pot za vsakega agenta posebej, kot da ni drugih agentov, in poskuša odpraviti konflikte med temi potmi. Do konflikta pride, če sta dva agenta sočasno na istem mestu ali če agent zaseda prostor, ki ga mora drugi agent še prečkati. Pojem detekcija neodvisnosti pride iz tega, da algoritem išče poti na tak način, da se ob istem času dva agenta nikoli ne nahajata na isti točki. Če konfliktov ne more odpraviti uporabi dekompozicijo operatorjev. Dekompozicija operatorjev je pristop k večagentnemu iskanju, kjer se položaje skupine agentov tretira kot eno stanje sveta. Za primerjavo algoritmov sem implementiral še preprost algoritem, ki poišče eno pot, poišče poti od začetnih položajev agentov do najbližjih točk na poti in simulira premikanje.

Zgeneriral sem scenarije, kjer en scenarij predstavlja seznam parov začetkov in ciljev. Vsak algoritem sem pognal na scenarijih in simuliral izvajanje premikanja agentov. Izmeril sem čas iskanja poti, razpršenost skupine agentov in druge metrike in zbral ter ustrezno predstavil rezultate.

1.2 Strateške igre

Strateške igre so računalniške ali druge igre za dva ali več igralcev. Izid igre je odvisen od odločitev, ki jih spremajo igralci tekom igranja igre. Obstaja več vrst strateških iger: potezne (npr. Civilization), namizne (npr. šah), itn. Za to diplomsko delo so zanimive predvsem realno časovne računalniške strateške igre (Real-Time Strategy oz. RTS). Primera takih iger sta Starcraft in Command&Conquer.

Cilj realno časovne strateške igre je, da igralec ali zavezništvo bodisi uniči nasprotnika ali preživi do časovne omejitve, nakar se zmago dodeli tistemu, ki ima več sredstev, enot. Igralec doseže zmago, tako da smotrno dodeli sredstva, ki jih ima na razpolago, za gradnjo stavb in izdelavo enot. Z izdelanimi enotami se nato lahko brani ali napada. Velik del igre je sestavljen iz nadzora nad skupinami enot. Od skupine se pričakuje, da zna pri znanem cilju v

čim krajšem času in na čimboljši način poiskati pot po bojišču. Potrebno je ločiti med enoto strateške igre, pod čemer si lahko predstavljamo kmeta na šahovnici ali računalniškega avatarja, ki ga lahko uporabnik izbere in z njim manipulira od agenta, s katerim operirajo algoritmi te diplomske naloge. Agent je zgolj par položaja na bojišču in identiteta, brez vizualne predstavitve, medtem ko ima enota tipično bistveno več parametrov (velikost, posebne zmožnosti, izkušnje, itd.) in je v računalniških igrah dobro vizualno predstavljena.

Bojišče v strateških igrah je ponavadi pravokotna mreža polj, kjer je vsako polje lahko bodisi prehodno ali neprehodno. Ta bojišča so tipično zgrajena, tako da se skupine enot lahko premikajo kot skupine. To pomeni, da je malo ozkih prehodov (mostovi, predori, itd.), ki so kljub majhnosti dovolj široki, da jih lahko prečka več agentov hkrati. Bojišče lahko razdelimo na več s prehodi povezanih površin, pri čemer je vsaka taka površina večinoma prehodna za skupino agentov.

1.3 Povzetek rezultatov

Po implementaciji osnovnih algoritmov se je izkazalo, da obstaja veliko možnosti za izboljšave. Prva večja izboljšava je bila nadomestitev algoritma A* z algoritmom HPA*, kar močno pospeši iskanje poti za posameznega agenta. Hitrost iskanja poti za enega samega agenta je ozko grlo vseh implementiranih algoritmov za večagentno iskanje. Druga večja izboljšava se nanaša na večagentno iskanje poti z uporabo hodnika. Predvidljivo obnašanje sistema delcev je v praksi težko doseči, zato sem namesto pristopa s silami uporabil kar vektorsko polje, ki teče vzdolž glavne poti, s čemer sem izničil oscilacije agentov okoli glavne poti in enotnost skupine. Tretja izboljšava se nanaša na algoritem z detekcijo neodvisnosti in dekompozicijo operatorjev. Algoritem je zaradi dekompozicije operatorjev in ponavljajočega se iskanja poti za posamične agente zelo dolgo iskal poti. Z odstranitvijo dekompozicije operatorjev in pospešitvijo generiranja poti za več agentov iz ene glavne poti

sem pohitril delovanje algoritma.

Uporabljena algoritma v osnovni obliki nista učinkovito iskala poti na domeni strateških iger. Izkazalo se je, da ne ohranjata enotnosti skupine in delujeta prepočasno za uporabo v realnem času. Z izboljšavami, ki sem jih implementiral, sem pokazal, da se algoritma lahko prilagodi za uporabo v realnem času. Čas iskanja poti se da drastično izboljšati z uporabo algoritma HPA*, odpravo dekompozicije operatorjev in odpravo iskanja poti za vsakega agenta posebej. Enotnost skupine agentov se s pravilnim pristopom lahko ohrani, tako da se skupina nikoli ne razdeli in agentje ne obstanejo na mestu.

Struktura diplomskega dela:

- V Poglavju 2 so opisi vseh algoritmov za večagentno iskanje (VaIP z uporabo hodnika, VaIP z detekcijo neodvisnosti in dekompozicijo operatorjev, algoritmov A* in HPA*
- V Poglavju 3 je opis poteka implementacje algoritmov, težav s katerimi sem se srečal in kako sem jih odpravil
- V Poglavju 4 je opis bojišč, na katerih sem testiral algoritme, in omejitve gibanja agentov, potek testiranja (število poskusov, strojna oprema), rezultati in interpretacija rezultatov

Poglavje 2

Algoritmi

2.1 A*

Algoritem A* [3] je znan hevristični algoritem. Spada v družino t.i. informiranih algoritmov. Razlika med iskanjem v globino in A* je hevristika, ki jo algoritem uporablja za prioritizacijo vozlišč, ki jih razvija. Tipični hevristiki sta Manhattanska in Evklidova razdalja.

Funkcijo, ki vodi preiskovanje, v splošnem definiramo kot [1]:

$$f(n) = g(n) + h(n)$$

kjer je:

- $h(n)$ hevristična ocena razdalje med vozliščem n in ciljnim vozliščem
- $g(n)$ razdalja od začetnega vozlišča do vozlišča n

Algoritem najde optimalno rešitev, v primeru da je hevristika popolna (ang. admissible). Za vsako vozlišče n definirajmo $h^*(n)$, ki izračuna ceno optimalne poti od vozlišča n do ciljnega vozlišča. Hevristika je popolna, če velja:

$$h(n) \leq h^*(n)$$

Ocena mora biti torej vedno manjša ali enaka dejanski ceni poti.

Algorithm 1 Algoritem A*

```
closedset ← {}
openset ← {start}
predecessor ← {}
g[start] = 0
f[start] = g[start] + heuristic(start, goal)
while not openset.empty() do currentNode ← openset.pop()
    if currentNode == goalNode then
        return reconstructed path
    end if
    remove currentNode from openset
    add currentNode to closedset
    for neighbour in currentNode.neighbours do
        ng ← g[currentNode] + movecost
        if neighbour ∈ openset AND ng < g[neighbour] then
            remove neighbour from openset
        else if neighbour ∈ closedset AND ng < g[neighbour] then
            remove neighbour from closedset
        else if neighbour not in openset OR ng < g[neighbour] then
            predecessor[neighbour] = currentNode
            g[neighbour] = ng
            f[neighbour] = ng + heuristic(neighbour, goal)
            if neighbour not in openset then
                add neighbour to openset
            end if
        end if
    end for
end while
return no solution
```

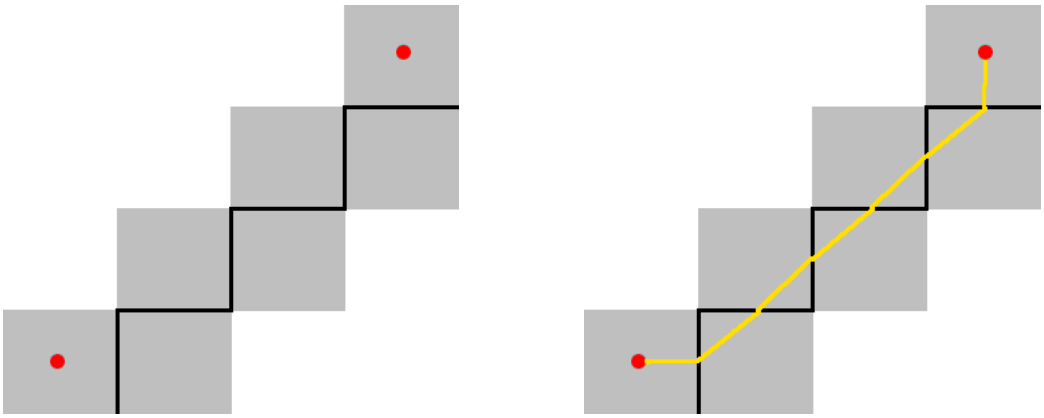
2.1.1 Zgoščene vrednosti

Pomemben del algoritma A* je ugotavljanje identičnosti dveh stanj. Če sta stanji identični ju lahko tretiramo kot eno in isto stanje. Problem pri primerjanju dveh stanj nastane, kadar je stanje kompleksnejša podatkovna struktura (npr. stanje, ki vsebuje več agentov z dodeljenimi položaji). V tem primeru se uporabi zgoščeno vrednost (ang. hash function). Namesto primerjave vseh sestavnih delov stanja se primerja zgolj vrednosti izračunanih zgoščenih vrednosti. Posledica uporabe zgoščene funkcije, je da se lahko dve različni stanji preslikata v isto vrednost. Za pravilno delovanje je potrebno zagotoviti, da se to ne zgodi. Uporabil sem dve zgoščeni vrednosti. Za točke zemljevida p sem uporabil preprosto formulo $hsh = p.x * 10000 + p.y$. Uporabljena bojišča so velikosti do 9999x9999, zato je taka zgoščena funkcija dovolj dobra. Za stanja z več agenti in položaji sem uporabil algoritem MD5 [10] nad vsemi položaji agentov, ki so del stanja. Položaje agentov se pretvori v zaporedja znakov, ki se jih združi v eno dolgo zaporedje znakov w. Zgoščeno vrednost dobimo iz niza w, tako da nad njim poženemo algoritem MD5.

2.2 HPA*

Algoritem HPA* [4] je izboljšava algoritma A*. Algoritem predhodno izdela višjenivojski model bojišča, tako da bojišče predprocesira. Tekom predprocesiranja se iz bojišča ustvari abstrakten graf, ki omogoča višjenivojsko preiskovanje prostora. Algoritem poišče pot v abstraktnem grafu in jo nato izostri, da dobi uporabno pot sestavljeno iz točk bojišča.

Bojišče razdeli na gruče (ang. cluster). Gruča je v tem diplomskem delu omejena s pravokotnikom določene dolžine in širine in jo sestavljajo prehodi, ki potekajo po robovih pravokotnika. Vsak prehod je vozlišče v abstraktnem grafu. Prehod med dvema vozliščema v abstraktnem grafu je največji segment roba, ki ne vključuje nobene ovire in poteka po stranici enega pravokotnika in najbližji stranici sosednjega pravokotnika.



(a) Primer poti v abstraktnem grafu (b) Izostrena pot.

Slika 2.1: HPA* najprej poišče pot v abstraktnem grafu in jo nato izostri. Črni robovi označujejo vozlišča. Rdeči piki označujeta začetno in končno vozlišče.

Definirajmo:

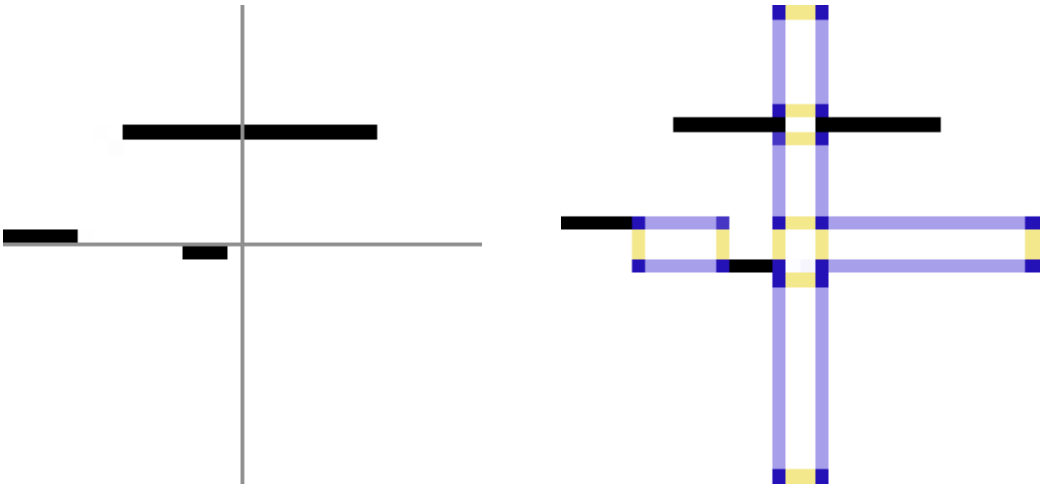
- c_1 in c_2 sta dva soležna pravokotnika
- l_1 in l_2 sta roba pravokotnikov c_1 in c_2, ki se stikata horizontalno ali vertikalno
- symm(t) je funkcija, ki točko $t \subset l_1 \cup l_2$ preslika v točko $t' \subset l_1 \cup l_2$ in velja, da sta točki t in t' simetrični glede na premico, ki je vzporedna in poteka med roboma l_1 in l_2

Prehod e mora izpolnjevati naslednje pogoje:

- $e \subset l_1 \cup l_2$
- $\forall t \in l_1 \cup l_2 : t \in e \iff symm(t) \in e$
- Prehod ne vsebuje ovir.
- Prehod je neprekinjen segment vzdolž roba, ki se začne z začetkom roba ali točko za oviro in konča s koncem roba ali točko pred oviro. En rob lahko vsebuje več prehodov.

Vsak prehod je vozlišče v abstraktnem grafu (ang. inter-edge). Prehodu v eni gruči vedno ustreza prehod v drugi gruči. V vsaki gruči poiščemo najkrajšo pot med vsakim parom robov (ang. intra-edge) in shranimo bodisi dolžino poti bodisi celotno pot. V primeru, da shranimo samo dolžino poti, mora algoritem HPA* vsakič znova iskati poti med notranjimi Te podatke uporablja algoritem HPA* med iskanjem rešitve. Algoritem HPA* lahko pri preiskovanju uporablja katerokoli točko prehoda, zato se pojavi vprašanje, katera točka je najbolj optimalna. V tem diplomskem delu sem problem poenostavil, tako da se upošteva zgolj točko na sredini prehoda, s čemer sem želel poiskati poti, ki so dobre za skupino agentov(okoli sredinske točke je praviloma največ praznega prostora). Izkazalo se je, da to ni optimalna izbira, vendar v povprečju deluje dovolj dobro. Statična informacija o terenu je v strateških igrah praviloma znana. V primeru, da se teren spremeni

(recimo da nasprotnik postavi stavbo v ozek prehod), je potrebno lokalno predprocesirati spremenjen del bojišča pred ponovno uporabo.



(a) Bojišče s prikazanim terenom

(b) Robovi na bojišču so označeni z modro. Meja roba je označena z rumeno.

Slika 2.2: Primer bojišča velikosti 32x32, s pravokotnikom velikosti 16x16

Abstraktni graf se uporabi tako, da algoritem poišče pot od gruče, ki vsebuje začetno točko, do gruče s končno točko. Rezultat je pot v abstraktnem grafu. Algoritem pridobi konkretno pot iz abstraktne, tako da jo izostri in hkrati zgladi. Postopek zajema:

- Iskanje začetne poti, ki poteka od začetne točke do prvega vozlišča v abstraktnem grafu(sredina roba gruče, v kateri se nahaja začetna točka)
- Iskanje in izostritev poti v abstraktnem grafu.
- Iskanje končne poti, ki poteka od začetne točke do prvega vozlišča v abstraktnem grafu(sredina roba gruče, v kateri se nahaja začetna točka)

Pri iskanju začetne poti je potrebno izbrati tudi prvo vozlišče v abstraktnem grafu, zato da se lahko poišče pot v abstraktnem grafu. V diplomskem delu sem dodal vsa vozlišča gruče, v kateri se nahaja začetna točka, kot vhod v algoritem A*, ki preiskuje abstraktni graf.

Velikost pravokotnika, ki omejuje posamezno gručo, vpliva na čas iskanja poti. Večji kot je pravokotnik, več časa algoritem išče začetno in končno pot. Po drugi strani večji pravokotnik pomeni manj gruč in predvidoma manj robov, kar pohitri iskanje poti v abstraktnem grafu. V primeru večje velikosti pravokotnikov traja iskanje začetne in končne poti dlje časa. Po drugi strani izbira manjše velikosti pravokotnika poveča prostor stanj za abstraktni graf in poveča čas iskanja v abstraktnem grafu.

Glajenje poti se lahko implementira s preprosto prilagoditvijo algoritma A*. Namesto preiskovanja po dvodimenzionalni mreži se najprej preveri, če obstaja pot naravnost do cilja. Na strateškem zemljevidu, razdeljenem na pravokotnike, je veliko pravokotnikov brez ovir, kar pomeni, da se da preiskovanju s pomočjo algoritma A* marsikdaj izogniti. S tem se fazo predprocesiranja močno pospeši.

Prednosti algoritma HPA* sta nizka kompleksnost implementacije in pospešitev hitrosti preiskovanja. Hitrost preiskovanja je ključna pri uporabi v realno časovni strateški igri, kjer se od algoritmov pričakuje učinkovitost in odzivnost. Po drugi strani ima algoritem HPA* kar nekaj slabosti. Algoritem HPA* ni primeren za vse tipe bojišč. Preprost primer, kjer se ne obnese dobro, je bojišče, naključno posejano z neprehodnim terenom. Problem, ki se pojavi pri taki vrsti bojišča, je ogromna količina robov, kar povzroči izjemno dolgo predprocesiranje in večjo količino informacije, ki jo je potrebno shranjevati in nalagati. Še ena slabost je, da algoritem HPA* ne najde optimalnih poti. V tem diplomskem delu nisem testiral, kako blizu optimalnim potem je HPA*, vendar se da iz vizualiziranih testov razbrati, da odstopanje ni veliko.

Algoritem HPA* je dobro prilagojen za bojišča, kjer se neprehodne točke bojišča ne spreminjajo. V realno časovnih strateških igrah se več agentov tipično ne more nahajati na isti poziciji, kar pomeni da so nasprotnikovi agentje stavbe ovire, podobno kot neprehoden teren. Za uporabo v taki igri je algoritem HPA* potrebno prilagoditi, tako da lokalno predprocesira del zemljevida, ki postane neprehoden zaradi nestatičnih ovir.

Poseben problem algoritma HPA*, ki nastane v povezavi z drugimi algo-

ritmi (VaIP z uporabo hodnika), je koncept minimalnega prostega prostora okoli poti. Algoritem VaIP z uporabo hodnika in VaIP z detekcijo neodvisnosti in dekompozicijo operatorjev morata dobiti poti, po in okoli katerih lahko razporedita skupino agentov. Zato mora faza predprocesiranja algoritma HPA* upoštevati minimalen radij praznega prostora okoli iskanih poti. Navedeno se lahko implementira, tako da se algoritmu za predprocesiranju poda primeren minimalen radij, ki ga morajo imeti vse točke na prehodih in vse poti med vozlišči gruče, skozi katere algoritem za predprocesiranje išče pot. To pomeni, da bosta omenjena algoritma za VaIP delovala učinkovito le do določenega minimalnega radija in za določene velikosti skupine agentov.

Algoritem HPA* omogoča uporabo večnivojske hierarhije, kjer je najnižji nivo bojišče v izvorni obliki, nivo višje abstraktno bojišče z velikostjo gruče mxn. Abstraktnemu bojišču, razdeljenem na gruče velikosti mxn, lahko sledi še več še bolj abstraktnih bojišč, kjer je velikost gruče vsakič večja od nižjega nivoja. Izkazalo se je, da uporaba večnivojske hierarhije ne pomeni tudi dviga hitrosti preiskovanja. Pospešitev se pozna samo od določene razdalje preiskovane poti naprej. Uvedba je smiselna, če se skupina agentov pogosto premika čez večji del bojišča.

Algorithm 2 Algoritem HPA*

```
naivePath = framedastar(graphs.last.graph, start, goal)
if naivePath ≠ None then          ▷ Pot v primeru bližine začetka in cilja
    return naivePath
end if
currPath = []
revGraphs = graphs.reversed ▷ Grafi od najbolj do najmanj abstraktnega
for rect, graph in revGraphs do
    if currPath ≠ None then
        currPath = abstractastar(graph, start, goal, currPath, rect)
    end if
end for
if currPath == None then
    return None       ▷ V primeru da ni rešitve v abstraktnem grafu tudi
konkretne rešitve ni
end if
abstractStart = halfedgepoint(currPath[0])
startPath = framedastar(graphs.last.graph, start, abstractStart)
currentGraph = graphs.last.graph
path = startPath
for i in 1..len(currPath) do
    currentStartPoint = halfEdgePoint(currPath[i-1])
    currentEndPoint = halfEdgePoint(currPath[i])
    midPath = framedastar(currentGraph, currentStartPoint, currentEnd-
Point)
    path += midPath
end for
abstractGoal = halfedgepoint(currPath[len(currPath)-1])
endPath = framedastar(graphs.last.graph, abstractGoal, goal)
path += endPath
return path
```

2.3 Preprosto večagentno iskanje poti

Preprosto večagentno iskanje je poenostavitev iskanja poti za več agentov. Namenjen je primerjavi z izboljšanimi algoritmi. Algoritem poišče glavno pot od začetne do končne točke prvega agenta. Za vsakega agenta posebej s pomočjo algoritma HPA* poišče najbližjo pot do obstoječe poti. Agenti se nato premikajo ne glede na druge agente. V primeru, da se agent poskuša premakniti na že zasedeno polje se premik prekliče in agent stoji. Algoritem je uporaben za testiranje in primerjanje z ostalimi algoritmi, saj deluje podobno, kot tipičen algoritem v realno časovnih strateških igrah, ki agente tipično postavi v gosji red, kadar sta začetna in končna točka dovolj narazen.

2.4 Večagentno iskanje poti z uporabo hodnika

Algoritem iskanja poti z uporabo hodnika [5] opisuje postopek, po katerem skupina agentov doseže cilj. Agenti se morajo izogibati trkom med seboj in ohranjati enotnost, tako da se posamezni agenti drug od drugega ne oddaljijo preveč. Enotnost se doseže, tako da se poišče glavno pot (ang. backbone path), katere se držijo vsi agenti. To pot se poišče z algoritmom kot je HPA*. Na tej poti se definira točke, ki jim agenti sledijo (ang. attraction points). Okoli te glavne poti definiramo hodnik, v katerem se lahko agenti gibajo. S tem, da omejimo razdaljo med točkami privlačnosti, dosežemo, da se skupina obnaša enotno. Delovanje algoritma lahko razdelimo na dva dela. Prvi del je iskanje glavne poti. Drugi del je nadzor gibanja posameznih agentov znotraj hodnika okoli glavne poti.

2.4.1 Iskanje glavne poti

Pri iskanju glavne poti je pomembno predvsem, da se jo najde hitro, ker je od hitrosti iskanja odvisen čas do začetka premikanja agentov proti cilju in da ima vsaka točka na poti za nek minimalen radij r prosto prehodnega polj

okoli sebe. Večji kot je ta prostor, več svobode imajo agenti pri premikanju znotraj hodnika. Po drugi strani lahko izbiranje določenih poti, kjer ima večina točk poti večji radij prosto prehodnega prostora okoli sebe, vodi k izbiranje suboptimalnih poti.

Eden izmed problemov, na katerega sem naletel pri tem algoritmu, je omejenost HPA*, ki ne upošteva omejitve minimalnega praznega prostora okoli sebe. Algoritem HPA* določi dve točki, ki sta del roba (ekstremni na vsaki strani), za vozlišči, ki nastaneta, ko rob najdemo. Moja implementacija to poenostavi in vzame najprimernejšo točko roba. Točka mora imeti v radiju r zgolj prehodne prostor. Če take točke ni, se med suboptimalnimi točkami roba poišče najprimernejši glede na to, koliko prehodnih točk ima točka roba v okolici..

2.4.2 Nadzor agentov

Gibanje posameznih agentov znotraj poti poteka na osnovni enostavnih pravil. Od skupine se pričakuje, da se obnaša kot realna skupina oseb, ki se držijo skupaj tako vzdolž smeri gibanja (longitudinalno), kot pravokotno na smer gibanja (lateralno). Nadzor nad gibanjem se doseže tako, da se vpelje konstanti longitudinalna in lateralna disperzija, s katerima se nadzira razpršenost skupine.

Glavna pot Π je pot po bojišču. Okoli vsake točke glavne poti obstaja krožnica $\rho_c(s)$ z radijem R, v kateri se lahko nahaja skupina agentov ali del skupine agentov.

Lateralna disperzija ob času t $D_{lat}(t)$ je maksimum vseh minimalnih razdalj od agenta U_i do agentu najbližje točke na glavni poti $\Pi(s)$.

$$D_{lat}(t) = \max_i \Big\{ \min_{s\in[0,1]} \Big\{ \|\Pi(s) - U_i(t)\| \Big\} \Big\}$$

Točke atrakcije AP_i so točke, ki jim agenti skupine lahko sledijo. Točke atrakcije jih praviloma vodijo bližje cilju ali bližje skupini agentov. Formalno se jih definira kot:

$$AP_i(t) = \Big\{ s \in [0,1] : U_i(t) \in \rho_c(s) \Big\}$$

Največjo lateralno disperzija je enostavno dobiti iz lokacij agentov in glavne poti. Posledično jo algoritem lahko tudi omejuje s pravilnim nastavljanjem sil. Longitudinalno razdaljo se po drugi strani lahko dobi na več načinov. Avtorja sta predlagala sledečo definicijo:
Najmanjšo in največjo skupinsko točko atrakcije ($s^b(t)$ in $s^f(t)$) se dobi iz najmanjše in največje točke atrakcije za posameznega agenta U_i:

$$s_i^b(t) = minAP_i(t)$$

$$s_i^f(t) = maxAP_i(t)$$

Najmanjša skupinska točka atrakcije je definirana kot:

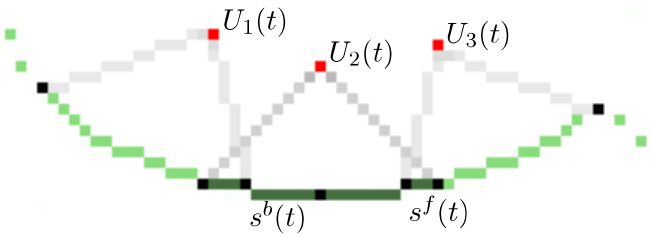
$$s^b(t) = \min_i \Big\{ s_i^f(t) \Big\}$$

Največja skupinska točka atrakcije je definirana kot:

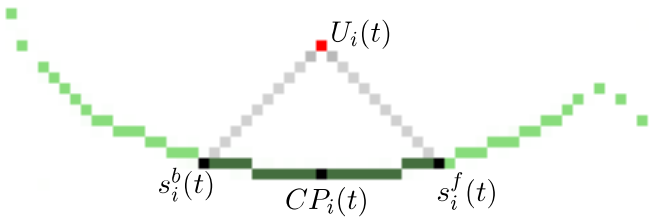
$$s^f(t) = max\Big(\max_i \Big\{ s_i^b(t) \Big\}, s_b(t)\Big)$$

Longitudinalna disperzija je razdalja med maksimalno in minimalno skupinsko točko atrakcije vzdolž glavne poti.

$$D_{long}(t) = dist_{\Pi}\Big(s^b(t), s^f(t)\Big)$$

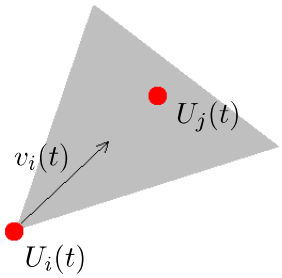


Slika 2.3: Najmanjša ($s^b(t)$ ali $s_1^f(t)$) in največja ($s^f(t)$ ali s_3^b) skupinska točka atrakcije.



Slika 2.4: Možne točke atrakcije na poti za agenta, ki ga ponazarja rdeč kvadrat. $CP_i(t)$ je točka na glavni poti, ki je najbližje agentu. Formalno je definirana kot $CP_i(t) = \underset{s\in[0,1]}{argmin}\Big\{\|\Pi(s) - U_i(t)\|\Big\}$

Agenti sledijo točkam atrakcije zaradi sil, ki so definirane na tak način, da usmerjajo agente v pravo smer. Trki znotraj skupine so nezaželeni, zato se definira tudi odbojno silo med agenti. Hitrost in položaje se dobi tako, da se vzame vse sile posameznih agentov in uporabi numerične metode za reševanje nastalega sistema enačb. Sila, ki vleče agente vzdolž poti, vedno kaže v točko na glavni poti. Sila kaže čimdlje vzdolž poti vendar ne dlje od maksimalna skupinske točke atrakcije. Odbojna sila deluje med agenti tako, da en agent izpodriva drugega, kadar sta preblizu.



Slika 2.5: Odbojna sila $v_i(t)$ vpliva samo na agenta $U_j(t)$, ki je na poti agentu $U_i(t)$

Algoritem deluje, tako da vsakemu delcu nastavi silo in uporabi numerične metodo Runge-Kutta četrtega reda za izračun položajev in hitrosti delcev [2]. Če problem začetne vrednosti definiramo kot:

$$\dot{y} = f(t, y), y(t_0) = y_0$$

Kjer je y znana funkcija, za katero se išče vrednost odvoda in so znane začetne vrednosti. Za izbrani časovni korak $h > 0$ definirajmo:

$$y_{n+1} = y_n + 1/6(k_1 + 2k_2 + 2k_3 + k_4)$$

$$t_{n+1} = t_n + h$$

za $n = 0, 1, 2, ...$ in velja:

$$k_1 = f(t_n, y_n)$$
$$k_2 = f(t_n + \frac{1}{2}h, y_n + h\frac{1}{2}k_1)$$
$$k_3 = f(t_n + \frac{1}{2}h, y_n + h\frac{1}{2}k_2)$$
$$k_4 = f(t_n + h, y_n + hk_3)$$

Algorithm 3 Runge Kutta metode I

```
function updateParticles(p, dt, iterations, updateForces, path, lateralDispersion, longitudinalDispersion)
    deriv1 = initialState(p)
    p1tmp = calculateDerivatives(p, deriv1, 0)
    updateForces(p1tmp, path, lateralDispersion, longitudinalDispersion)
    deriv2 = initialState(p1tmp)
    p2tmp = calculateDerivatives(p1tmp, deriv2, dt * 0.5)
    updateForces(p2tmp, path, lateralDispersion, longitudinalDispersion)
    deriv3 = initialState(p2tmp)
    p3tmp = calculateDerivatives(p2tmp, deriv3, dt * 0.5)
    updateForces(p3tmp, path, lateralDispersion, longitudinalDispersion)
    deriv4 = initialState(p3tmp)
    for i in range(len(p)) do
        p[i].p = dt * (deriv1[i].dpdt / 6 + deriv2[i].dpdt / 3 + deriv3[i].dpdt
/ 3 + deriv4[i].dpdt / 6)
        p[i].v = dt * (deriv1[i].dvdt / 6 + deriv2[i].dvdt / 3 + deriv3[i].dvdt
/ 3 + deriv4[i].dvdt / 6)
    end for
end function
```

Algorithm 4 Runge Kutta metode II

```
function initialState(p)
    deriv = []
    i = 0
    for particle in p do
        deriv[i].dpdt = particle.v
        deriv[i].dvdt = particle.f / particle.m
        i += 1
    end for
    return deriv
end function
function calculateDerivatives(initialP, deriv, dt)
    ret = initialP.clone()
    for i in range(len(ret)) do
        ret[i].p = initialP[i].p + deriv[i].dpdt * dt
        ret[i].v = initialP[i].v + deriv[i].dvdt * dt
    end for
end function
```

2.5 Večagentno iskanje poti z detekcijo neodvisnosti in dekompozicijo operatorjev

Algoritem Večagentno iskanje poti z detekcijo neodvisnosti in dekompozicije operatorjev [6, 7] je prilagoditev algoritma A* za večagentno iskanje poti. Temelji na dveh principih. Detekcija neodvisnosti (ang. independence detection) in dekompozicija operatorjev (ang. operator decomposition). Jedro detekcije neodvisnosti je tabela konfliktov. Algoritem uporabi algoritem A*, tako da poišče pot za vsakega agenta posebej. Te poti se nato vnese v podatkovno strukturo(tabelo konfliktov). V primeru, da ni konfliktov, je algoritem že našel pot za skupino agentov. Če konflikti ob določenem času obstajajo, potem algoritem najprej ponovno splanira pot za prvega agenta

pri čemer se izogne poti drugega. Če algoritmu to ne uspe, ponovno splanira pot za drugega agenta, pri čemer se izogne poti prvega. Oboje lahko spodleti in v tem primeru, detekcija neodvisnosti ne vrne rešitve. V primeru da detekcija neodvisnosti ne vrne rešitve se problem večagentnega iskanje rešuje z dekompozicijo operatorjev. Pri detekciji neodvisnosti je pomembno, da so nekatere dobljene poti lahko krajše od drugih, kar pomeni da je potrebno paziti, da ni konfliktov med daljšimi potmi in zadnjo točko krajše poti. Algoritem v osnovi ne zagotavlja, da se agenti dejansko premikajo kot skupina, vendar je testiranje pokazalo, da se agenti premikajo kot skupina. Enotnost skupine je odvisna od števila agentov. Več kot je agentov, slabša je enotnost.

Do konflikta med dvema agentoma lahko pride v treh primerih:

- ob času t_0 imata obe poti p_1 in p_2 enak položaj: $p_1[t_0] = p_2[t_0]$
- ob času t_0 se pot p_1 zaključi na nekem položaju $p_1[t_0]$ in obstaja pot, ki je daljša od p_1 in vsebuje isti položaj ob času t_1
- ob časih t_0 in t_1 in potema p_1 in p_2 velja $p_1[t_0] = p_2[t_1]$ in $p_1[t_1] = p_2[t_0]$

V primeru, da detekcija neodvisnosti ne reši problema, se združi agente, ki imajo konfliktne poti, v eno skupino. Za to skupino se rešitev išče globalno, pri čemer eno stanje prostora stanj obsega dodeljene položaje oz. premike (!) posameznemu agentu skupine (dekompozicija operatorjev). Stanja delimo na dva tipa. Standardno stanje je stanje, kjer ima vsak agent položaj in noben agent nima dodeljenega naslednjega položaja. Vmesno stanje je stanje, kjer ima vsaj en agent dodeljen naslednji položaj, na katerega se lahko premakne(to je vseh osem koordinat zraven trenutnega agentovega položaja in agentov položaj - agent lahko stoji). Začetno stanje je standardno stanje, kjer je vsakemu agentu dodeljen začetni položaj. Začetno stanje ima 9 sosednjih vmesnih stanj, ki predstavljajo možne premike enega agenta. Sosedi vmesnega stanja so vmesna stanja, ki agentu brez dodeljenega naslednjega položaja dodelijo možne naslednje položaje. Ko vmesno stanje dodeli naslednji položaj zadnjemu agentu nastane novo standardno stanje

tako, da naslednji položaji postanejo trenutni agentovi položaji. Rešitev, ki jo najde algoritem, vsebuje ponavljajoča se zaporedja standardnih in vmesnih stanj. Ker so vmesna stanja redundantna jih algoritem izloči, tako da rezultat vsebuje samo standardna stanja.

Medtem ko je detekcija neodvisnosti časovno še sprejemljiva za uporabo v realno časovni strategiji je dekompozicija operatorjev časovno bistveno bolj potratna in se ji je smiselno za vsako ceno izogniti. V najslabšem primeru lahko dekompozicija operatorjev združi vse agente v eno samo skupino, kar praviloma rezultira v kombinatorični eksploziji in je časovno neobvladljivo.

Algorithm 5 Detekcija neodvisnosti in dekompozicija operatorjev

```
conflicts = PathConflictTable()
solutions = []
conflictsAttempted = []
for i in range(startPos) do
    solution = odastar(graph, [ startPos[i] ], [ goals[i] ])
    conflicts.addPath(solution)
    solutions.append(list(solution))
end for
c = conflicts.getConflict()
while c ≠ None do
    resolved = False
    firstGroup is first group of units in c
    secondGroup is second group of units in c
    if c not in conflictsAttempted then
        possibleSolution = astar(graph, firstGroup, firstGoal, conflicts, c[1])
        if possibleSolution ≠ None then
            resolved = True
            solutions[firstGroup] = possibleSolution
        end if
    end if
    if (c[1], c[0]) not in conflictsAttempted AND NOT resolved then
        possibleSolution = odastar(graph, secondGroup, secondGoal, con-
flicts, c[0])
        if possibleSolution ≠ None then
            resolved = True
            solutions[secondGroup] = possibleSolution
        end if
    end if
    if NOT resolved then
        newGroup = group joined from group1 and group2
        newGoals = concatenated goals from group1 and group2
        solution = odastar(graph, newGroup, newGoals)
    end if
    recreate conflicts table
    c = conflicts.getConflict()
end while
```

Poglavje 3

Implementacija algoritmov

3.1 Opis implementacije algoritmov

V jeziku Python sem najprej implementiral algoritem iskanja poti z uporabo hodnika. Sprva je algoritem deloval zgolj na majhnih bojiščih velikosti od 10x10 do 20x20. Ko sem ga pognal na bojiščih, ki se dejansko uporabljajo v strateških igrah (velikost 512x512 in več) se je algoritem A* izkazal kot prepočasen. Posledično sem namesto A* uporabil HPA*, kar je močno pospešilo proces iskanja glavne poti.

Algoritem HPA* je sestavljen iz dveh delov. Prvi del je predprocesiranje zemljevida. Izdelal sem skripto, ki procesira bojišča, dosegljiva na http://www.movingai.com/benchmarks/sc1/. Vsa bojišča so shranjena v preprostem tekstovnem formatu. Podatki se naložijo v instanco razreda v Python-u, ki se jo shrani in nalaga v datoteke s pomočjo modula cPickle. Python-ov mehanizem za serializacijo podatkov se je izkazal, saj ni bilo potrebno pisati dodatno kodo za shranjevanje in nalaganje podatkov v datoteke.

Podobno skripto sem napisal za generiranje scenarijev. Scenarij je seznam, kjer je vsak element par seznamov točk, ki predstavljajo začetne in končne položaje agentov. Podobno kot pri procesiranju bojišč za HPA* sem uporabil modul cPickle za shranjevanje in nalaganje scenarijev. Drugi del al-

goritma HPA* je prilagoditev A*, ki poišče pot v abstraktnem grafu. Rezultat iskanja v abstraktnem grafu je zaporedje pravokotnikov bojišča, skozi katere poteka pot. Algoritem HPA* nato izostri pot s pomočjo informacij predprocesiranega bojišča.

Algoritem iskanja poti z uporabo hodnika se je izkazal kot težaven za implementacijo predvsem zaradi podrobnosti v implementaciji sil. Sile sem sprva definiral točno po opisu algoritma v članku [5] vendar so agenti močno oscilirali okoli poti. Do oscilacij je prihajalo zaradi računanja položajev z metodo Runge Kutta 4. reda. Zaradi nestabilnosti sem uporabil drug pristop. Definiral sem silo, ki kaže vzdolž poti, zaviralno silo in stabilizacijsko silo. Sila vzdolž poti potiska agenta naprej vzdolž poti (tangentno na pot). Zaviralna sila preprečuje, da agent preskakuje polja zaradi visoke hitrosti. Sila je definirana kot $F = ma$ pri čemer je $m = 1$. Konstantna sila delcu stalno povečuje hitrost, zato je omejevanje hitrosti nujno potrebno. Stabilizacijska sila preprečuje osciliranje okoli poti. Do osciliranje pride zato, ker je posameznemu delcu razmeroma težko določiti silo, tako da se giblje vzdolž poti v tirnici vzporedni s potjo.

Osciliranja kljub precej poskusom nisem onemogočil. Omejil sem obseg oscilacij, ki se s časom zmanjšajo na zanemarljivo raven. Do večjih oscilacij pride vedno v predelih, kjer pot močno spremeni smer. Algoritem sem izboljšal z odstranitvijo sistema delcev, ki deluje na osnovi sil. Namesto sistema delcev sem vpeljal vektorsko polje, ki delce izven hodnika usmerja v hodnik, delce v hodniku nato usmerja vzdolž poti. Delec v vektorskem polju se lahko premakne na tri položaje. Vzdolž poti ali vzdolž diagonalno glede na pot. Izboljšani algoritem izbere tisti položaj, ki ni zaseden.

Standleyev algoritem [6, 7] sem ravno tako najprej implementiral na manjšem prostoru. Ko sem ga pognal na večjih bojiščih se je izkazalo, da ima precejšnje probleme s hitrostjo iskanja posamezne poti. Za razliko od iskanja z uporabo hodnika ta algoritem poišče pot za vsakega agenta posebej, kar pomeni precejšnjo izgubo časa pri iskanju posameznih poti. Na manjših bojiščih z velikim številom agentov išče pot globalno, kar močno poveča pros-

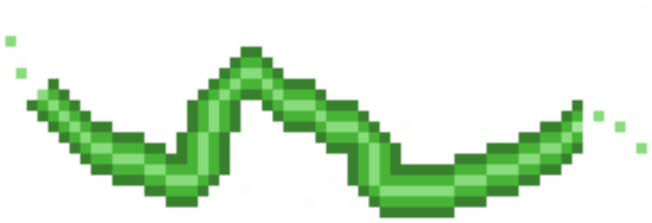
tor stanj in ne deluje več v realnem času. Algoritem sem izboljšal tako, da sem uporabil hodnik kot pri VaIP z uporabo hodnika in odstranil dekompozicijo operatorjev.

Izboljšani algoritem poišče hodnik od začetka enega agenta do cilja, nato znotraj hodnika zgenerira poti za več agentov. Algoritem ustvari poti, tako da ob glavni poti izdela vzporedne poti. Koliko vzporednih poti lahko ustvari, je odvisno od širine hodnika. V primeru konfliktov med posameznimi potmi, jih algoritem poskuša odpraviti z iskanjem alternativnih poti za posamezne agente. Konflikti lahko nastanejo na začetku ob iskanju poti od začetnih položajev agentov do pripadajoče poti. Če ne najde alternativnih poti, algoritem ne vrne rešitve. Algoritem agente enakomerno razporedi na posamezne poti, zato da je skupina čimbolj enotna.

Pri obeh izboljšanih algoritmih se je izkazalo, da se skupina agentov še vedno razprši. Pri večjih skupinah se skupina razdeli na tri dele. Glavnina agentov se praviloma drži skupaj (glavnino sestavlja polovica do tri četrtine agentov). Poleg glavnine se agentje razdelijo na tiste, ki zaostajajo zaradi neenakovrednih začetnih položajev in agente, ki ubežijo glavnini, ker jih ni nič oviralo. Problem sem rešil tako, da sem vpeljal periodično regrupiranje. Na vsakih n premikov agentje, ki so najdlje na poti, čakajo k premikov agente, ki zaostajajo. Za vsakega agenta izračunam razdaljo l_{long} vzdolž poti od agenta, ki najbolj zaostaja. Vsak agent, za katerega velja $l_{long} > c$, kjer je c longitudinalna razdalja, ki jo dobi program kot parameter, čaka na mestu.

Še en problem izboljšanih algoritmov in osnovnega VaIP je postavitev skupine na koncu simulacije poti. Od algoritmov se pričakuje, da bodo skupino agentov spravili na cilj, pri čemer bo skupina na cilju še vedno enotna. Problem je torej prehod skupine agentov iz hodnika ali glavne poti, kjer so agentje bolj ali manj razpršeni, v krožnico okoli cilja skupine g. Ta problem sem rešil na podoben način za vse algoritme. Na začetku iskanja poti algoritmi izdelajo diskretni krog točk radija r. Ko agent s položajem p_i pride dovolj blizu cilja g, torej velja $|p_i - g| < r$, se agentu dodeli končni položaj, tako da se izbere točko iz diskretnega kroga c_j, se jo prišteje cilju g,

da se dobi končni cilj za agenta $g_i = c_j + g$. Agentu se ob izpolnitvi pogoja $|p_i - g| < r$ spremeni pot, tako da napreduje v ravni črti do dodeljenega cilja g_i.



Slika 3.1: Vzporedne poti ob glavni poti (najsvetlejša črta). V tem primeru je minimalna širina hodnika 5.

Za primerjavo algoritmov sem nato implementiral trivialen algoritem (preprosto večagentno iskanje), ki zgolj poišče glavno pot, poišče poti od agentov do najbližje točke na poti in simulira premikanje. Statistike, ki jih dobim na osnovi tega algoritma, služijo za primerjavo in razlago, v kakšnem obsegu sta izboljšana algoritma boljša od preprostega algoritma.

Izboljšana algoritma sta se izkazala pri testiranju na majhnem številu agentov. Pri testiranju na več primerih in z več agenti je prišlo do precejšnjega upada uspešno doseženih ciljev. Razlog je bil v tem, da hodnik pogosto ni dosegal minimalne širine. Na segmentih poti so se agenti ustavili zaradi neprehodnega terena, ki je bil preblizu glavni poti. VaIP z detekcijo neodvisnosti in dekompozicijo operatorjev sem razširil z algoritmom, ki združuje zunanje poti z notranjimi v primeru, da pot vsebuje ovire. VaIP z uporabo hodnika sem izboljšal tako, da se posamezni agentje ne morejo več premakniti izven dejanskega hodnika. Pred tem so se lahko premikali kjerkoli v radiju r okoli glavne poti. Rezultat obeh izboljšav je močno povečanje uspešno poiskanih poti.

Ohranjanje enotnosti večjega števila agentov se je izkazalo kot težavno. Večina agentov se praviloma drži skupaj. Manjšina agentov bodisi prehiteva

(a) Zožanje hodnika pri VaIP z detekcijo neodvisnosti. Z rjavo barvo so označeni deli najnižje poti, ki se na segmentih zlije s srednjo in glavno potjo.

(b) Zožanje hodnika pri VaIP z uporabo hodnika.

Slika 3.2: Zožanje hodnika pri izboljšanih algoritmih

bodisi zaostaja. Agente, ki zaostajajo, se z večjim delom agentov lahko združi, tako da se periodično preverja, če je razkorak med zadnjim agentom in trenutnim večji od neke konstante. V primeru, da je razkorak prevelik, agenti, ki so bližje cilju, čakajo zaostajajoče agente. Prehitre agente se združuje z glavno skupino tako, da preverjajo svojo okolico. V primeru, da ni v bližini nobenega agenta (VaIP z uporabo hodnika), ali da v okolici agenta na poti, ki ji sledi, ni nobenega agenta (VaIP z detekcijo neodvisnosti in dekompozicijo operatorjev), agent čaka.

3.2 Povzetek implementacije algoritmov

- Uvedba algoritma HPA* namesto algoritma A*
- Uporaba drugačnih sil (VaIP z uporabo hodnika)
- Nadomestitev sistema delcev s preprostim vektorskim poljem (VaIP z uporabo hodnika)
- Uporaba hodnika (VaIP z det. neodvisnosti in dek. operatorjev)
- Odstranitev dekompozicije operatorjev (VaIP z det. neodvisnosti in dek. operatorjev)
- Generiranje več vzporednih poti iz glavne poti (VaIP z det. neodvisnosti)
- Združevanje poti na zožanjih (VaIP z det. neodvisnosti)
- Periodično regrupiranje
- Čakanje agentov v ospredju

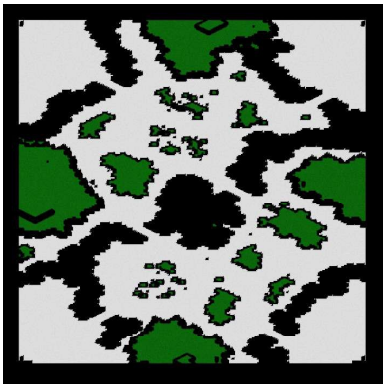
Poglavje 4

Poskusi

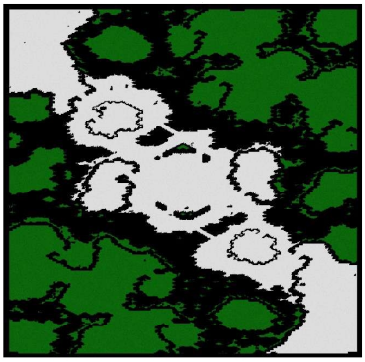
4.1 Domene

Domena za preiskovanje je MxN mreža polj. Velja naslednje:

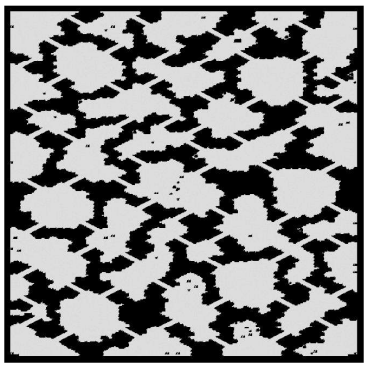
- vsako polje je lahko prehodno ali neprehodno
- na posameznem polju se lahko nahaja zgolj en agent
- vsako polje ima 8 sosedov (po dva horizontalno, vertikalno in diagonalno)
- agent se lahko premakne na 8 sosednjih polj ali stoji pri miru



(a) WarpGates - velikost 512x512



(b) CrashSites - velikost 768x768



(c) FloodedPlains - velikost 768x768

Slika 4.1: Bela barva označuje prehoden teren, medtem ko črna in zelena označujeta neprehoden teren.

4.2 Potek poskusov

Določil sem serijo metrik, s katerimi sem izmeril učinkovitost algoritmov:

- Uspeh - Uspeh je odstotek rešenih problemov (razmerje med uspešno rešenimi problemi in vsemi problemi).
- Povp. skupinski potovalni čas ($\bar{t}_g$)- Povprečen skupinski potovalni čas je povprečje števila korakov, da vsi agenti dosežejo cilj. Iz te vrednosti se lahko dobi povprečen čas, potreben za dosego cilja tako, da se jo množi s časovno konstanto.
- Čas izračuna ($\bar{t}_{calc}$) - Čas izračuna je povprečen čas za računanje poti v sekundah.
- Povp. največja razdalja med agenti ($\overline{dist}_{max}$) - Povprečna največja razdalja med agenti je povprečje največjih razdalj med enotami.
  Ta metrika meri razpršenost skupine v povprečju globalno.
- Povp. razdalja med agenti ($\overline{dist}$) - Za izračun povprečne razdalje med agenti se izračuna centroid (težišče položajev agentov v skupini) in razdaljo vsakega agenta do centroida. Povprečna razdalja med agenti je povprečje razlik položajev agentov in skupinskega centroida.
  Ta metrika meri razpršenost skupine lokalno.
- Pov. napadalna moč 1-n - Povprečna napadalna moč 1-n je metrika, ki opisuje napadalno moč. Na vsakem koraku simulacije algoritma se izračuna število polj v dosegu enega agenta, dveh agentov, itn. Po zaključku simulacije se izračuna povprečje. V strateških igrah je možnost, da več enot sočasno napada eno nasprotnikovo enoto, zaželjena, saj na ta način igralec lahko zmanjša izgube lastnih enot. Algoritmi poskušajo minimizirati manjše pov. napadalne moči(1-n/2) in ohranjati višje napadalne moči(n/2-n) čim višje. Metrika bolje ponazarja kompaktnost skupine kot druge že izračunane metrike. Metrika se izračuna tako, da se za vsakega agenta označi polja bojišča v krogu fiksne dolžine okoli

tega agenta. Vsako polje je lahko večkrat označeno. Na koncu vsakega koraka simulacije se prešteje koliko polj je bilo označenih enkrat, koliko dvakrat itn. Število enkrat označenih polj, dvakrat označenih polj itd. predstavlja napadalno moč 1, 2, itn. Za velikost skupine n je lahko eno polje označeno največ n-krat. Na koncu simulacije se vsako napadalno moč povpreči po vseh korakih simulacije. Konstanta, ki določa velikost kroga, za katerega se računa napadalno moč, je bila v testih fiksno postavljena na 4.

Povprečja sem sprva računal kot aritmetična povprečja, vendar se je izkazalo, da aritmetično povprečje ni najboljša metrika. V primeru, da za problem algoritem ni našel rešitve, so odstopanja v izmerjenih metrikah močno vplivala na rezultat. Posledično se iz rezultatov ni dalo razbrati, kako dober je algoritem v povprečju. Po zamenjavi aritmetičnega povprečja z mediano se je vpliv ekstremnih primerov močno zmanjšal.

Testiral sem na procesorju AMD Turion ZM82 na operacijskem sistemu Debian 'Squeeze', jedro 2.6.32-5-amd64. Vsako uporabljeno bojišče sem predprocesiral, zato da se ga lahko procesira z algoritmom HPA*. Za vsako bojišče sem zgeneriral 10 začetnih točk in cilj iskanja, pri čemer so evklidske razdalje med začetki in koncem enake ali malo odstopajo. Testiral sem s 5 agenti. Teste sem pognal neizboljšanimi in izboljšanimi algoritmi. Razlog, zakaj sem testiral zgolj s 5 agenti in 10 primeri, je predvsem zaradi počasnosti VaIP z detekcijo neodvisnosti in dekompozicijo operatorjev. Dekompozicija operatorjev močno vpliva na učinkovitost iskanja rešitev. Za bolj natančno predstavo o algoritmih sem stestiral še izboljšana algoritma na več primerih in agentih (100 primerov in skupine agentov velikosti 10,30 in 50).

Skupina doseže cilj, kadar se vsi agenti nahajajo v bližini cilja. Naj bo cilj g in c število agentov. Za vsakega agenta $i = 0..n$ s položajem p_i mora veljati

$$|p_i - g| < 2 * \sqrt{c}$$

Razlogi zakaj sem izbral samo en položaj za celotno skupino, namesto da bi vzel cilj za vsakega agenta posebej, so sledeči:

- Poudarek v tem diplomskem delu je na večagentnem iskanju poti. Pomembno je predvsem, da skupina doseže cilj.
- VaIP z uporabo hodnika v osnovi nima mehanizmov za natančno določanje, kako bo agent prišel na točno določen cilj. Cilj algoritma je, da skupina doseže neko destinacijo in ne vsak posamezen agent.
- VaIP z detekcijo neodvisnosti in dekompozicijo operatorjev lahko deluje tako, da ima vsak agent skupine specifičen cilj, vendar rešitve ne najde v predvidljivem času. Lahko pride do kombinatorične eksplozije, kar povzroči dolgotrajno iskanje rešitve.
- V realno časovni strateški igri se cilji enot pogosto spreminjajo. Problema natančnih postavitev enot tako pogosto niti ni potrebno reševati.

4.3 Rezultati

Imena algoritmov v tabeli so skrajšana z namenom lažje berljivosti.

1. *Base* je osnovni test namenjen preverjanju izboljšav
2. *Clear* je algoritem iskanja z uporabo hodnika
3. *ID+OD* je algoritem iskanja z detekcijo neodvisnosti in dekompozicijo operatorjev
4. *Clear** je izboljšana različica algoritma iskanja z uporabo hodnika
5. *ID+OD** je izboljšana različica algoritma iskanja z detekcijo neodvisnosti in dekompozicijo operatorjev

4.4 Primerjava osnovnih in izboljšanih algoritmov

Algo.	Uspeh	$\overline{t}_g$	$\pm$	$\overline{t}_{calc}[s]$	$\pm[s]$	$\overline{dist}_{max}$	$\overline{dist}$	1	5
Base	100	172.0	9.0	7.5	2.2	12.9	4.6	57.4	1.7
Clear	100	233.0	18.0	2.5	0.0	11.1	3.7	90.4	0.8
ID+OD	80	161.0	3.0	43.8	25.9	7.8	2.5	52.0	11.9
Clear*	100	188.0	17.0	1.8	0.0	3.6	1.3	29.1	32.3
ID+OD*	100	173.0	7.0	2.1	0.1	3.6	1.3	34.1	28.5

Tabela 4.1: Bojišče WarpGates; 10 primerov; 5 agentov

Algo.	Uspeh	$\overline{t}_g$	$\pm$	$\overline{t}_{calc}[s]$	$\pm[s]$	$\overline{dist}_{max}$	$\overline{dist}$	1	5
Base	100	267.0	48.0	9.0	1.0	9.3	2.9	64.6	0.1
Clear	100	350.0	79.0	2.8	2.8	13.5	4.6	118.5	0.3
ID+OD	40	216.0	39.0	213.1	193.5	4.0	1.4	50.4	10.6
Clear*	100	275.0	38.0	2.1	1.5	3.2	1.2	24.2	35.1
ID+OD*	100	253.0	41.0	2.3	1.9	3.4	1.2	32.9	28.5

Tabela 4.2: Bojišče CrashSites; 10 primerov; 5 agentov

Algo.	Uspeh	$\overline{t}_g$	$\pm$	$\overline{t}_{calc}[s]$	$\pm[s]$	$\overline{dist}_{max}$	$\overline{dist}$	1	5
Base	100	253.0	23.0	9.1	2.5	8.6	2.9	45.8	2.2
Clear	100	327.0	41.0	3.9	0.6	13.9	4.3	99.2	0.4
ID+OD	40	232.0	30.0	386.8	253.4	4.1	1.2	37.5	19.1
Clear*	100	290.0	25.0	2.5	0.2	3.2	1.2	25.6	34.5
ID+OD*	100	251.0	21.0	2.7	0.5	2.8	1.0	26.2	36.2

Tabela 4.3: Bojišče FloodedPlains; 10 primerov; 5 agentov

4.5 Testi izboljšanih algoritmov

Algo.	Uspeh	$\bar{t}_g$	$\pm$	$\bar{t}_{calc}[s]$	$\pm[s]$	$\overline{dist}_{max}$	$\overline{dist}$	1	5	10
Base	100	186.0	25.5	7.6	2.6	17.0	4.7	50.1	24.3	0.0
Vector*	98	220.5	33.0	4.1	0.7	6.7	2.0	28.6	11.7	15.3
ID+OD*	99	193.0	25.5	4.4	0.8	5.6	1.8	31.4	10.8	20.0

Tabela 4.4: Bojišče WarpGates; 100 primerov; 10 agentov

Algo.	Uspeh	$\bar{t}_g$	$\pm$	$\bar{t}_{calc}[s]$	$\pm[s]$	$\overline{dist}_{max}$	$\overline{dist}$	1	5	10
Base	100	192.5	21.5	10.7	5.7	37.1	9.4	41.8	55.7	2.3
Vector*	97	235.0	29.0	3.9	0.5	13.9	3.1	36.0	13.9	8.4
ID+OD*	100	191.5	22.5	4.9	0.7	13.0	2.9	34.4	13.9	8.4

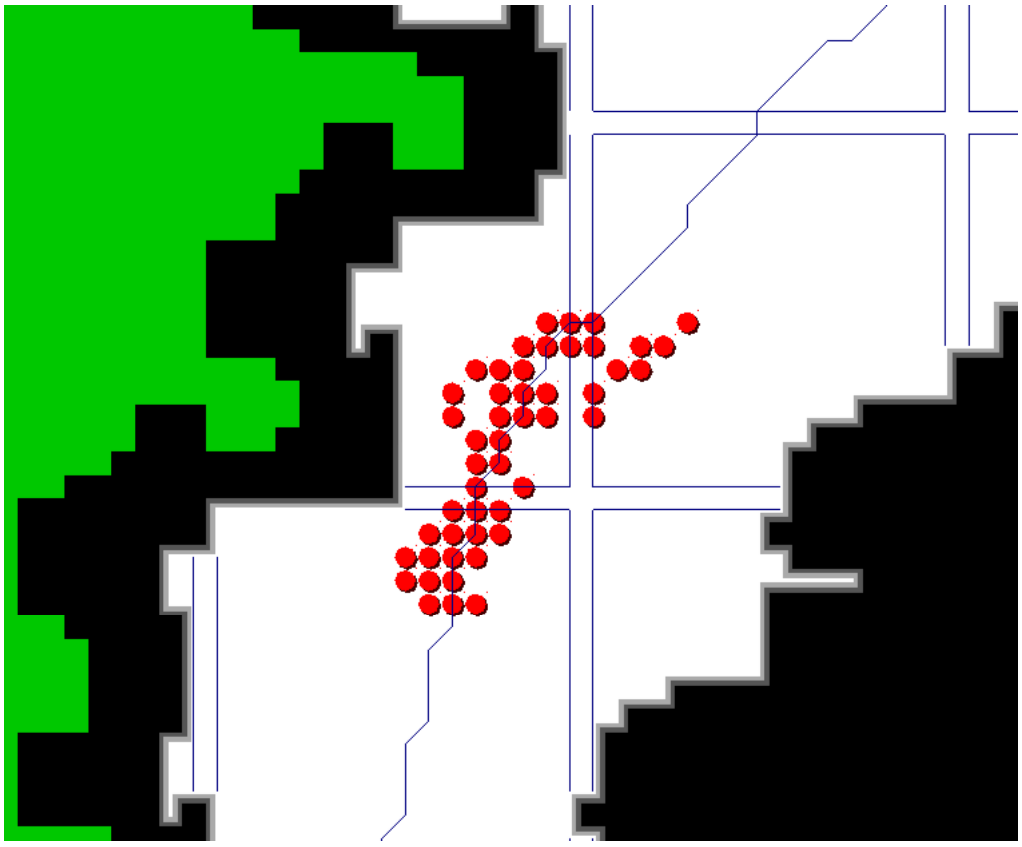
Tabela 4.5: Bojišče WarpGates; 100 primerov; 30 agentov

Algo.	Uspeh	$\bar{t}_g$	$\pm$	$\bar{t}_{calc}[s]$	$\pm[s]$	$\overline{dist}_{max}$	$\overline{dist}$	1	5	10
Base	96	212.5	21.0	18.3	12.5	52.4	13.6	46.5	78.1	4.4
Vector	93	261.0	39.0	3.8	0.7	20.7	4.4	39.2	17.5	10.9
ID+OD*	97	190.0	23.0	5.1	0.8	21.6	4.9	38.4	19.9	12.0

Tabela 4.6: Bojišče WarpGates; 100 primerov; 50 agentov

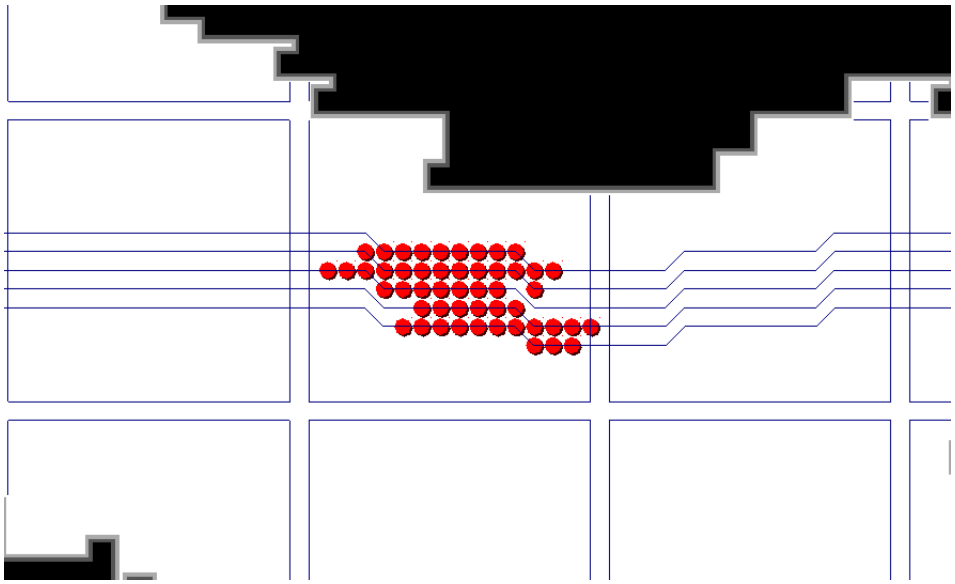
4.6 Ponazoritev delovanja algoritmov

(a) Osnovno VaIP

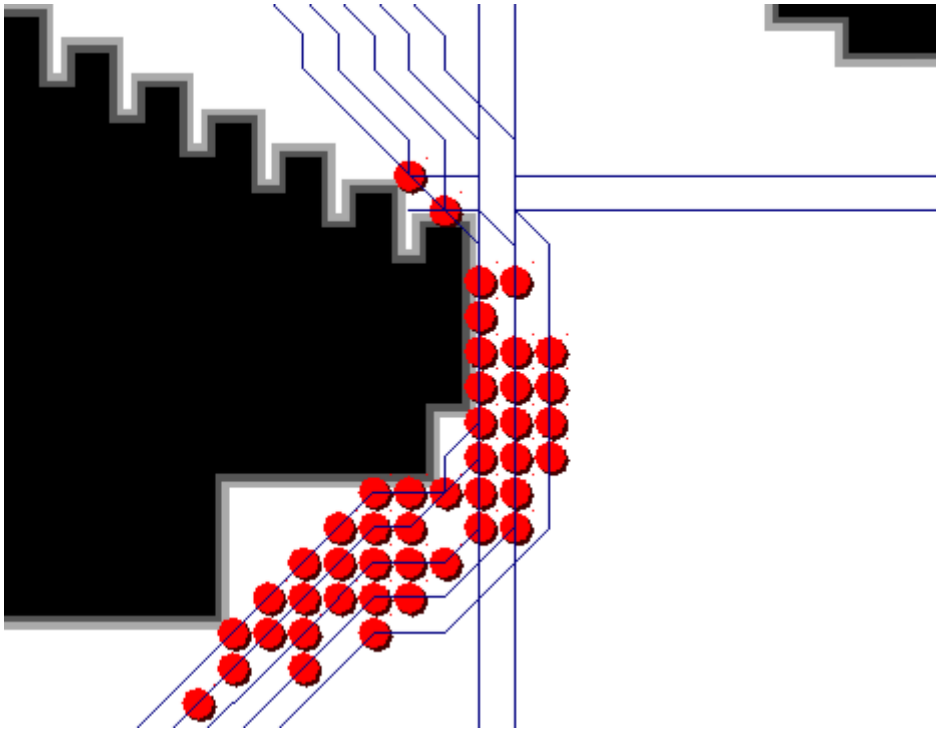


(b) VaIP z uporabo hodnika

Slika 4.2: Algoritmi med simuliranjem I



(a) VaIP z detekcijo neodvisnosti



(b) Zožanje hodnika pri VaIP z detekcijo neodvisnosti

Slika 4.3: Algoritmi med simuliranjem II

4.7 Interpretacija

Iz rezultatov primerjave osnovnih in izboljšanih algoritmov se vidi, da je čas računanja za osnovni ID+OD algoritem izjemno dolg. To omejuje uporabnost algoritma v realno časovni strateški igri, kjer se od algoritmov za iskanje poti pričakuje odzivnost. Algoritem najprej išče pot za vsakega agenta posebej in nato odpravi konflikte. Medtem ko se pot za vsakega agenta posebej še najde razmeroma hitro, odpravljanje konfliktov med potmi vzame mnogo več časa, ker se v ta namen uporablja algoritem z dekompozicijo operatorjev.

Kot kaže izboljšana različica algoritma, se da problem v večji meri odpraviti. Čas računanja je v testih izboljšanih algoritmov bistveno manjši. Algoritem se torej z določenimi spremembami da uporabiti v strateških igrah za večagentno iskanje. Te spremembe so odprava dekompozicije operatorjev in uporaba hodnika kot pri algoritmu za večagentno iskanje poti z uporabo hodnika.

Posledica uporabe hodnika je tudi, da je skupina v izboljšanem algoritmu enotnejša, kar se vidi iz metrike povprečna napadalna moč 1 in 5. Zmanjšanje napadalne moči 1 pomeni, da je manj polj, ki jih lahko napade zgolj en sam agent. Povečanje napadalne moči 5 pomeni, da lahko v povprečju več agento sočasno napade eno točko. V izboljšanem algoritmu je to število manjše, kar pomeni, da je v povprečju skupina agentov bolj kompaktna in se ne razprši. To pomeni, da se skupina agentov lahko brani in napada bolj učinkovito.

Izboljšave se da doseči tudi pri algoritmu iskanja z uporabo minimalne širine. Medtem ko pri tem algoritmu čas ni tak problem kot pri ID+OD algoritmu, je problem, da se agenti razpršijo, kar se vidi tako iz povprečne največje razdalje med agenti in povprečne napadalne moči 1. Obe metriki sta pri originalnem algoritmu precej visoki, zaradi načina usmerjanja agentov. Enote se tretira kot sistem delcev, ki se mu nastavi sile, nato algoritem za reševanje sistemov enačb preračuna položaje in hitrosti delcev z metodo Runge-Kutta 4. reda. Položaje se nato zaokroži na celo vrednost, zato da se lahko položaje delcev nastavi agentom v diskretni mreži.

Izboljšani algoritem namesto sistema za reševanje sistema enačb uporablja

bolj preprost algoritem. Delce usmerja v smer vzdolž glavne poti glede na velikost in smer sile. Ob vsakem časovnem koraku se vsakemu delcu izbere tri naslednje položaje vzdolž glavne poti. Delcu se določi tisti naslednji položaj, ki ni v konfliktu z nobenim drugim delcem. Delec se zaradi preverjanja, ali je mogoč naslednji položaj izven hodnika, ne more premakniti iz hodnika. Posledica delovanja izboljšanega algoritma je, da je skupina bolj enotna. Zaradi zadrževanja delcev v hodniku se zmanjša tudi povprečni potovalni čas.

Iz testov izboljšanih algoritmov se natančneje vidi karakteristike izboljšanih algoritmov. Večja skupine agentov vplivajo na uspešnost iskanja in simulacije poti. Pri 50 agentih število uspešno rešenih problemov rahlo upade glede na 10 in 30 agentov tako pri izboljšanem VaIP z uporabo hodnika kot izboljšanem VaIP z detekcijo neodvisnosti. Do neuspeha v večini primerov pride zaradi prekoračenja časovne omejitve (algoritem najde pot, vendar ne uspe vseh agentov premakniti dovolj blizu cilja). Izkazalo se je, da problem razporejanja velikega števila agentov iz hodnika na neko končno pozicijo ni trivialen.

Po drugi strani se oba izboljšana algoritma dobro obneseta pri metrikah povprečna razdalja med agenti, povprečna največja razdalja med agenti, povprečna napadalna moč 5 in 10 od osnovnega VaIP. Več kot je agentov, manjše so povprečne razdalje in večje so povprečne napadalne moči z višjim indeksom, kar pomeni da izboljšana algoritma dobro ohranjata enotnost skupine.

Poglavje 5

Zaključek

Rezultati kažejo, da se lahko učinkovitost izbranih algoritmov za večagentno iskanje močno poveča. Izboljšani algoritmi delujejo drastično hitreje in so že skoraj primerni za uporabo na realnem projektu. To se vidi iz padca časa za računanje poti, ki je upadel iz več 100 sekund na manj kot 10 sekund. Enotnost skupine je pri obeh izboljšanih algoritmih bistveno boljša. Metrika napadalna moč 1 je v določenih testih tudi za polovico manjša, medtem ko so vse višje napadalne moči večje, kar implicira, da je skupina bistveno bolj enotna. Opazno je tudi zmanjšanje povprečnega potovalnega časa za izboljšano VaIP z uporabo hodnika. Izboljšana algoritma ohranjata učinkovitost tudi pri večjih skupinah agentov in daljših poteh, kar se vidi iz dodatnih testov za izboljšane algoritme.

Obstaja še veliko možnosti za nadaljnje izboljšave. Glede na to, da je algoritem HPA* ozko grlo obeh algoritmov za VaIP bi bilo smiselno poiskati algoritem, ki deluje hitreje. Algoritmi ne upoštevajo, da se lahko agenti gibajo različno hitro ali da so lahko različnih velikosti. Zanimivo bi bilo tudi preveriti, kako topologija zemljevida vpliva na delovanje algoritmov. Mehanično razdeljena mreža je enostavna za implementacijo vendar ni optimalna, predvsem v zožanjih na zemljevidu. Izbrane metrike so v določenih situacijah lahko bolj ali manj koristne, česar v tem diplomskem delu nisem upošteval. Za konec bi bilo smiselno tudi avtomatizirat nastavljanje parametrov širina

hodnika, omejitev razpršenosti vzdolž poti, pogostost regrupiranja id., ki so zdaj fiksni in se jih nastavi na roke pri poganjanju algoritmov.

Literatura

[1] I. Bratko, *Prolog Programming for Artificial Intelligence*, 4. izdaja, Pearson, 2011.

[2] B. P. Flannery, W. H. Press, S. A. Teukolsky in W. Vetterling, *Numerical recipes in C.*, 2. izdaja, Press Syndicate of the University of Cambridge, 1992, str. 710-714.

[3] P. E. Hart, N. J. Nilsson in Bertram Raphael, *A formal basis for the heuristic determination of minimum cost paths.*, Systems Science and Cybernetics, IEEE Transactions on 4.2, 1968, str. 100-107.

[4] A. Botea, M. Müller in J. Schaeffer, *Near optimal hierarchical path-finding.*, Journal of game development 1.1, 2004, str. 7-28.

[5] A. Kamphuis in M. H. Overmars, *Finding paths for coherent groups using clearance.*, Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation, Eurographics Association, 2004, str. 19-28.

[6] T. Standley, *Independence Detection for Multi-Agent Pathfinding Problems.*, Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence, 2012.

[7] T. Standley, *Finding optimal solutions to cooperative pathfinding problems.*, Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10), 2010, str 173-178.

[8] D. Ratner in M. Warmuth, *Finding a shortest solution for the N×N extension of the 15-puzzle is intractable.*, Journal of Symbolic Computation 10, 1990, str. 111-137.

[9] M. Goldenberg et al, *A* Variants for Optimal Multi-Agent Pathfinding.*, Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence, 2012.

[10] R. Rivest, *RFC 1321: The MD5 message-digest algorithm*, The Internet Society, 1992, Status: INFORMATIONAl, http://tools.ietf.org/html/rfc1321.