Универзитет у Београду
Математички факултет

Никола Станојевић

Алгоритми за проналажење најкраћег пута у видео играма

Мастер рад

Ментор: **Миодраг Живковић**

Београд,

2017.

Ментор:

др Миодраг Живковић
Математички факултет
Универзитет у Београду

Чланови комисије:

др Предраг Јаничић
Математички факултет
Универзитет у Београду

мр Јелена Хаџи Пурић
Математички факултет
Универзитет у Београду

Датум одбране:

Садржај

1 Увод **1**

2 Начини представљања комплексних светова видео игара **3**
2.1 Графови за представљање мапа 3
2.2 Навигациони графови 4
2.2.1 Навигациони графови засновани на мрежи 4
2.2.2 Навигациони графови засновани на маркерима 6
2.2.3 Навигациони графови засновани на мрежама конвексних полигона 9

3 Захтеви и ограничења алгоритама за проналажење најкраћег пута у видео играма **11**
3.1 Проблем претраге навигационог графа 11
3.2 Захтеви алгоритама за проналажење најкраћег пута 13
3.2.1 Оптималност путање 14
3.2.2 Заобљавање путање 14
3.3 Ограничења алгоритама за проналажење најкраћег пута 16
3.3.1 Временска ограничења перформанси 17
3.3.2 Меморијска ограничења 18
3.4 Проналажење најкраћег пута у динамичком окружењу 20
3.5 Одабир одговарајућег алгоритма претраге 24

4 Најчешће коришћени алгоритми за проналажење најкраћег пута у видео играма **26**
4.1 Дискретни алгоритми претраге 27
4.1.1 Дијкстрин алгоритам 28
4.1.2 Алгоритам A* 38

5 Примери и тестови **49**
5.1 Примери начина представљања мапа 49
5.2 Поређење Дијкстриног алгоритма и алгоритма A* 51
5.2.1 Опис теста 51
5.2.2 Резултати теста 52
5.2.3 Анализа резултата теста 53
5.3 Пример динамичке промене мапе 54
5.3.1 Опис примера 54

6 Закључак **56**

Прилог: речник термина **60**

Библиографија **61**

Глава 1

Увод

Готово увек је потребно да карактерима у видео игри буде имплементиран одређени вид кретања по мапи. Понекад је кретање карактера на мапи предефинисано од стране онога ко развија видео игру. Такви видови кретања су на пример патролна путања коју ће чувар слепо пратити или ограђени простор којим ће се пас чувар насумично кретати. Овакве путање су унапред предвидљиве. Фиксиране путање су једноставне за имплементацију, али може настати много проблема ако се карактер помери са задате путање. Тако се слободни, лутајући карактер може лако заглавити у неком непредвиђеном делу мапе. Карактери код којих је кретање сложеније не знају унапред како ће изгледати њихова путања којом се крећу кроз окружење игре. На пример, карактери у стратегијама у реалном времену морају имати могућност слања од стране играча на било који део мапе.

Један од највећих изазова при изради видео игара јесте управо постизање реалистичног кретања карактера у видео игри. Стратегије проналажења најкраћег пута представљају језгро било ког система кретања имплементираног коришћењем вештачке интелигенције. Основни задатак ових стратегија јесте проналажење најкраћег или најбржег пута између било које две тачке у окружењу игре. У видео играма понекад најкраћи пут није и најбржи пут па се термин проналажења најкраћег пута често замењује термином планирање пута.

Систем за проналажење најкраћег пута је посредник између система за доношење одлука карактера у игри и система који омогућава његово кретање. Системи за проналажење најкраћег пута на основу почетне и крајње тачке

проналазе низ тачака које заједно чине најкраћу путању од почетне до крајње тачке. У процесу проналажења тачака путање користе се и одређене структуре података. У најједноставнијем облику то може бити листа локација које је могуће обићи на мапи. Проналажење најкраћег пута може бити веома скупа операција када се посматра искоришћеност ресурса рачунара, поготово ако се тражи проналажење путање која је непостојећа.

Велики број видео игара користи алгоритам A* за проналажење најкраћег пута. Алгоритам A* се највише користи због своје једноставности и ефикасности при имплементацији.

У глави 2 представљене су технике које омогућавају конвертовање комплексинх тродимензионалних мапа видео игара у навигационе графове који су једноставни за претрагу.

У глави 3 описани су захтеви и ограничења алгоритама за проналажење најкраћег пута у видео играма. Описано је какви проблеми могу настати приликом претраге навигационог графа и на који начин се ти проблеми могу превазићи. Проналажење најкраћег пута у динамичком окружењу видео игре је неопходно код већине данашњих видео игара. На крају главе описано је који алгоритам претраге је најбоље одабрати у зависности од проблема претраге који је потребно решити.

У глави 4 дат је преглед најчешће коришћених дискретних алгоритама за проналажење најкраћег пута у видео играма. Међу најпознатијим алгоритмима који спадају у ову групу алгоритама су Дијкстрин алгоритам и алгоритам A*. Приказан је опис наведена два алгоритма и разлике између њих.

У глави 5 приказани су примери и тестови реализовани коришћењем Јунити развојног окружења. Кроз пример је показана разлика између Дијкстриног алгоритма и алгоритма A*. Приказано је и прилагођавање ова два алгоритма динамичким променама на мапи, када се користе различити начини представљања мапа у видео играма.

У глави 6 дат је кратак закључак добијен на основу приказаних примера и тестова. Излозени су правци могуцег даљег рада у области. Излозени су правци могућег даљег рада у области проналажења најкраћег пута у видео играма.

Глава 2

Начини представљања комплексних светова видео игара

У данашњим видео играма мапе су веома сложене и у већини случајева тродимензионалне. Потребно је пронаћи што је могуће бољу и једноставнију апстракцију за представљање оваквог окружења. Најпогоднија структура података која се данас највише користи као апстракција комплексних светова видео игара је граф.

2.1 Графови за представљање мапа

Постоји више начина да се графом представе окружења која се користе у видео играма. Како би се омогућило коришћење алгоритама за проналажење најкраћег пута у видео играма, потребно је окружење игре представити структуром која је погодна за коришћење у алгоритмима. Неки од начина представљања равни у којој се игра одвија су помоћу плочица или полигона. У овом поглављу разматрају се неке од најпопуларнијих метода представљања равни таквим облицима и методе креирања графова који таквим поделама простора одговарају. Као најчешћа апстракција локација на мапи игре (које је могуће обићи) и свих веза између тих локација данас се највише користе навигациони графови [1],[2].

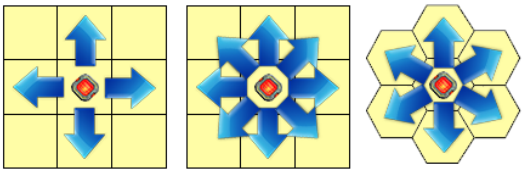
2.2 Навигациони графови

Навигациони граф представља упрошћени облик окружења у коме се игра одвија. Од начина апстракције датог окружења игре зависи величина и комплексност резултујућег навигационог графа. Добро конструисан навигациони граф омогућава обилажење свих битних путања и локација у окружењу игре и као такав веома је погодан за проналажење најкраћих путева између тих локација. Као и други графови, и навигациони графови се састоје од чворова и грана које те чворове међусобно повезују. Чворови навигационог графа обично представљају кључне области у окружењу игре, а свака грана представља везу између тих области. Гране навигационог графа могу имати додељене тежине којима се у најједноставнијем случају могу представити удаљености између чворова. Тежинама је могуће представити и одређне специфичне особине окружења игре које могу утицати на процес проналажења оптималног пута између две локације на мапи. Три најчешће коришћена облика навигационих графова су: навигациони графови засновани на мрежи, навигациони графови засновани на маркерима и навигациони графови засновани на мрежама конвексних многоуглова.

2.2.1 Навигациони графови засновани на мрежи

Навигациони графови засновани на мрежи се најчешће користе за представљање окружења RTS (енг. real time strategy, стратегије у реалном времену) игара и игара заснованих на ратовању неколико различитих фракција где се користе веома велике мапе. Чворови навигационог графа у RTS играма су најчешће плочице квадратног или хексагоналног облика. Када је реч о једноставним плочицама квадратног облика, некада је објектима у игри омогућено кретање само у четири различита правца, а може бити омогућено и кретање у осам праваца. Код хексагоналних плочица могуће је кретање у шест праваца, видети слику 2.1.

Када је мапа подељена на овај начин, има смисла навигациони граф посматрати на следећи начин: чворови су представљени централном тачком сваке од засебних плочица, а гране представљају везе између суседних ћелија.

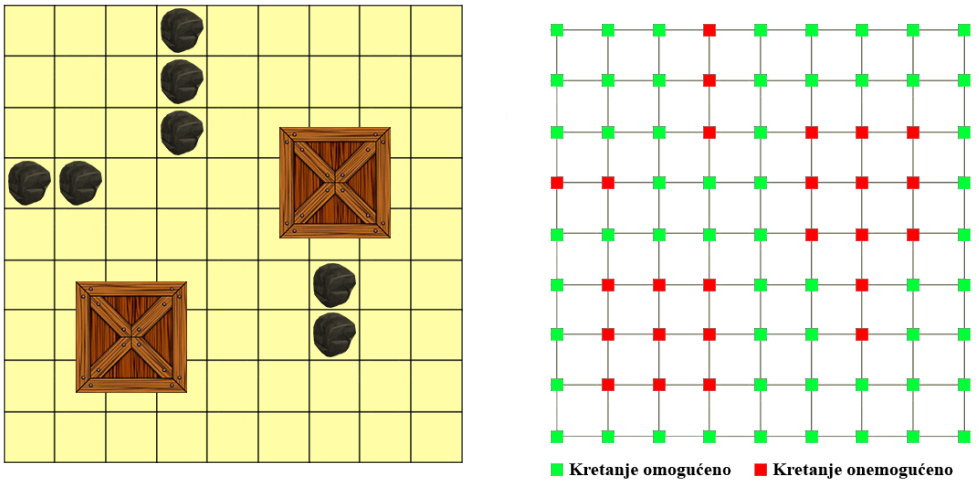


Слика 2.1: Плочице које омогућују кретање у четири, осам или шест праваца

Гранама је могуће доделити одређене тежине, које се касније могу користити у алгоритму за проналажење најкраћег пута. Тиме је омогућено проналажење најкраћих, или општије, најповољнијих путања. На пример, терен у игри може бити подељен на плочице различитих врста како би се представиле различите врсте терена (песак, вода, земља, блато). Уколико тенк треба да пређе путању на мапи од тачке А до тачке Б, потребно је узети у обзир преко какве врсте терена ће тенк прећи јер се он брже креће преко земље него преко воде. Проблем се решава тако што се додељују различите тежине гранама које воде преко воденог односно земљаног терена.

Важно је рећи да у овој репрезентацији могу постојати и чворови до којих објекти у видео игри не могу доћи, нпр. тенк се не може попети на неку веома високу планину. Сваки од чворова зато мора имати и индикатор који означава да ли се до њега може доћи или не. Дакле, навигациони граф заснован на мрежи садржи информације о томе које је чворове могуће, а које није могуће посетити, тј. њиме се може представити целокупно окружење игре. На слици 2.2 представљен је изглед навигационог графа заснованог на мрежи мапе која се састоји од квадрата, за које је омогућено кретање у четири различита правца.

Основна предност оваквог начина представљања мапа јесте у једноставном и ефикасном креирању навигационог графа. Све локације на мапи до којих се може и до којих се не може доћи представљене су чворовима. Добра особина навигационих графова заснованих на мрежи и јесте да се објекат који се креће по мапи и локација до које објекат треба да дође већ налазе на неком од чворова навигационог графа. То значи да није потребно вршити додатна израчунавања како би се пронашли њима најближи чворови навигационог графа. Још једна добра страна оваквог представљања мапа јесте лако прилагођавање навигационог графа променама на мапи једноставним изменама вредности индикатора чворова навигационог графа, тако да је веома користан у динамичким окружењима.



Слика 2.2: Мапа игре (лево) и њен одговарајући навигациони граф (десно)

Овакав начин представљања мапа има и своје лоше стране. Највећи проблем јесте то што простори претраге брзо постају превелики. Чак и за не претерано велике мапе димензија 100 пута 100 потребан је граф који има 10000 чворова и више од 78000 грана.

2.2.2 Навигациони графови засновани на маркерима

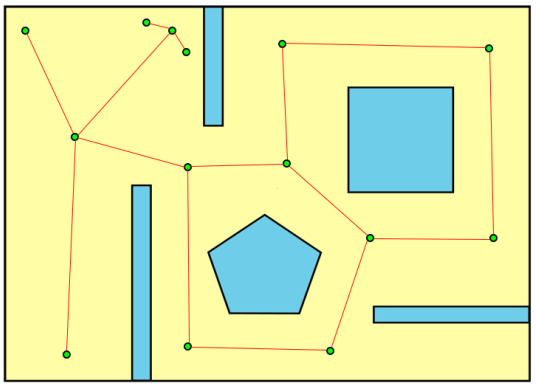
Један од традиционалних начина апстракције окружења у видео играма јесте помоћу навигационог графа заснованог на маркерима. За разлику од навигационог графа заснованог на мрежи, где је навигациони граф креиран на основу поделе окружења на плочице, највећи део посла око креирања навигационог графа заснованог на маркерима остављен је дизајнеру игре. Ако се пажљиво одаберу тачке на мапи могуће је добити навигациони граф који омогућава обилажење свих најзначајнијих области на мапи. Одабрани маркери истовремено представљају и чворове навигационог графа, који се затим повезују гранама ручно или аутоматски, како би се добио коначан изглед навигационог графа. Задатак дизајнера још у почетним фазама развоја игре јесте да постави маркере на мапи на основу којих ће се креирати навигациони граф.

Чворови (маркери) се повезују дужима које не пресецају ниједну препреку на мапи. Важно је напоменути да навигациони граф заснован на маркерима

за разлику од навигационог графа заснованог на мрежи не чува информацију о локацијама на којима се налазе препреке, већ садржи само чворове на локацијама које се могу обићи.

Као што је речено постављање маркера врши се ручно од стране дизајнера игре, али постоје и технике којима се маркери могу поставити аутоматски. Како је цена израчунавања локација маркера превелика, технике за аутоматизовање се ретко користе.

Навигациони графови засновани на маркерима не гарантују покривање целокупног простора по коме се објекти могу кретати на мапи, зато што може доћи до грешака услед људског фактора приликом постављања маркера. Уколико дизајнер игре изостави маркер, на мапи могу да се појаве недостижне локације, а уколико дизајнер постави превелики број маркера, може се непотребно увећати простор претраге. И један и други случај представљен је на слици 2.3. У области горе лево има сувишних маркера, а у области доле десно нема довољно маркера па је део мапе недостижан. Дакле, потребно је избалансирати између постављања премалог и превеликог броја маркера.



Слика 2.3: Ручно креиран навигациони граф заснован на маркерима

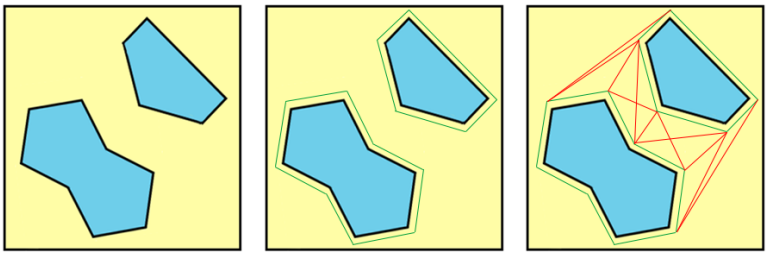
Једна од варијација навигационих графова заснованих на маркерима је POV (енг. points of visibility, заснован на тачкама видљивости) граф. Код POV графова чворови су такође повезани дужима које не пресецају препреке на мапи, али тако да сваки појединачни чвор мора бити повезан са свим другим њему видљивим чворовима. Добра страна POV графова јесте у томе што су лако проширриви, док је лоша страна то што, уколико је мапа веома велика, дизајнер може провести много времена како би направио и дотерао овакав граф. Још једна од лоших страна јесте проблематично коришћење POV графа уколико се користе самогенеришуће мапе. У том случају потребно је

аутоматизовати креирање графа заснованог на тачкама видљивости што није нимало једноставан задатак. Једно од решења овог проблема јесте коришћење техника проширене геометрије.

2.2.2.1 Технике проширене геометрије за генерисање POV графова

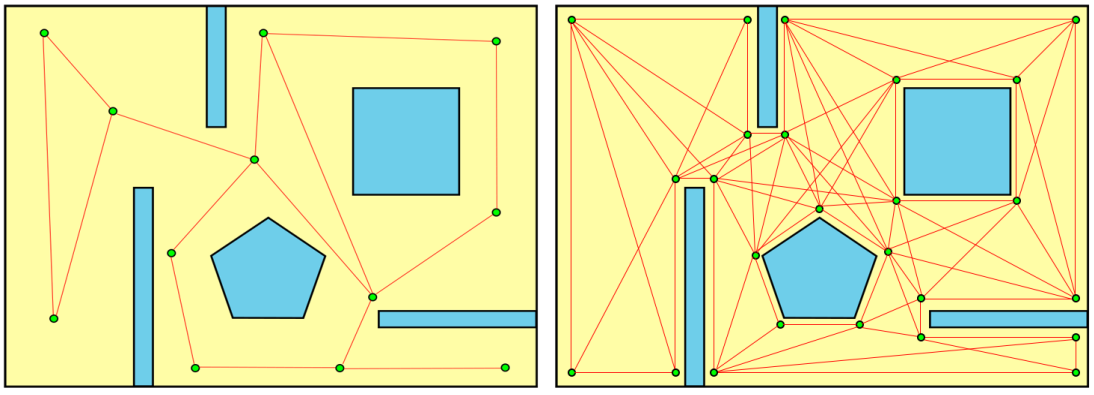
Уколико је окружење игре представљено полигонима могуће је из таквих облика извући одређене информације и на основу њих аутоматски конструисати POV граф, што може значајно да уштеди време дизајнеру игре.

Како би се постигла аутоматизација потребно је прво проширити границе објеката на мапи, видети слику 2.4. Чворови ПОВ графа затим постају темена која спајају ивице проширених објеката. На крају потребно је проверити између којих чворова се могу повући дужи тако да не пресецају ниједан објекат.



Слика 2.4: Процес проширивања објеката и генерисања навигационог графа

Добијене дужи су гране POV графа. На слици 2.5. представљено је поређење POV графа са ручно одабраним маркерима и POV графа добијеног коришћењем техника проширене геометрије.



Слика 2.5: POV граф са ручно одабраним маркерима (лево) и POV граф креиран коришћењем техника проширене геометрије (десно)

Добра страна POV графова добијених коришћењем техника проширене геометрије јесте што добијени навигациони граф сигурно покрива цео простор који се може обићи, што решава један од проблема ручно креираних навигационих графова. Међутим и даље остаје проблем редундантности чворова, поготово ако на мапи постоје кружни облици; тада може доћи до генерисања великог броја непотребних чворова. Дакле, са једне стране штеди се време дизајнера игре који сада не поставља маркере ручно, али са друге стране добија се навигациони граф који је доста комплекснији него што би могао да буде. Уколико је мапа велика, то значи да ће навигациони граф бити огроман и сложен у неким случајевима и потпуно неупотребљив. Могуће решење је да дизајнер ручно уклони редундантне чворове.

Оба метода заснована на маркерима и даље захтевају одређени степен интервенције дизајнера у процес генерисања навигационог графа. За огромне мапе потпуно је неприхватљиво користити претходне технике јер би процес креирања навигационог графа трајао превише дуго.

Претходне технике засноване на маркерима не гарантују потпуни квалитет (уклањање редундантних чворова није аутоматизовано) при одабиру маркера приликом промена у окружењу. Из тог разлога се навигациони графови засновани на маркерима користе само у статичним окружењима, врло ретко у динамичним.

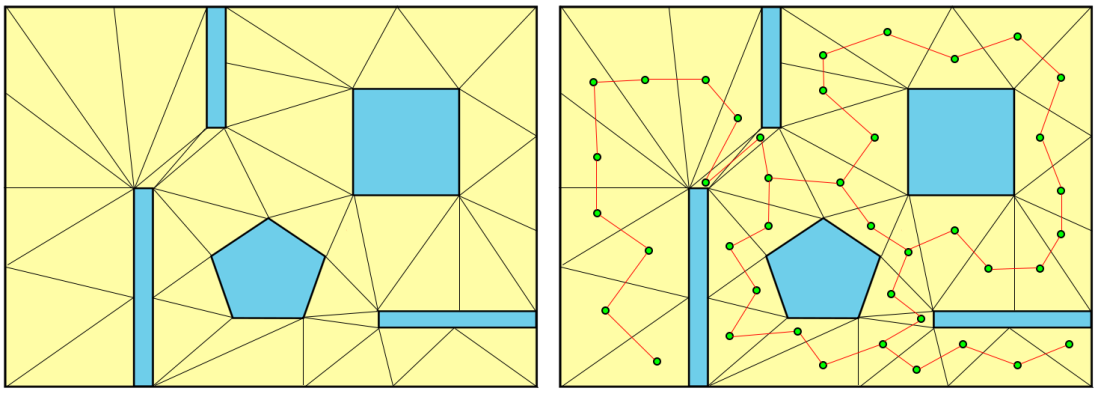
2.2.3 Навигациони графови засновани на мрежама конвексних полигона

Како би се омогућио високи степен аутоматизације приликом креирања навигационих графова у модерним играма користе се навигациони графови засновани на мрежама конвексних полигона. Коришћењем конвексних полигона за представљање простора на мапи који је могуће обићи минимизује се број чворова потребних у навигационом графу. Као и код навигационих графова заснованих на маркерима и навигациони графови засновани на мрежама конвексних полигона не садржи информацију о локацијама на којима се налазе препреке.

Конвексни полигони имају добро својство које омогућава неометано кретање између тачака полигона (сви унутрашњи углови су му мањи од 180^0 па

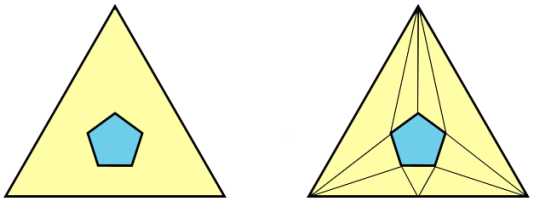
се ивице не могу пресецати). Претходно својство је корисно приликом представљања простора навигационим графом, јер чворови могу да представљају конвексни простор уместо само једне тачке у простору.

Често се као конвексни полигони за представљање мапа користе троуглови због најмањег броја грана који је потребан при таквој репрезентацији у навигационом графу. Троуглови су веома флексибилни при представљању комплексних окружења. Овакав начин представљања окружења зато пружа највернију реперезентацију простора који је могуће обићи. Коришћењем конвексних полигона добија се верна репрезентација простора који је могуће обићи. На слици 2.6. представљена је подела простора мапе на конвексне полигоне и одговарајући навигациони граф за дату поделу.



Слика 2.6: Мапа представљена помоћу конвексних полигона (лево) и њен одговарајући навигациони граф (десно)

Лоша страна је и даље коришћење поделе простора конвексним полигонима у динамичним окружењима, јер је потребно изнова прорачунавати поделу простора на полигоне што одузима време, видети слику 2.7.



Слика 2.7: Промена поделе на полигоне након додавања новог објекта на мапи

Свака промена у простору изазвала би промену поделе простора у области у којој је дошло до измена и промену навигационог графа.

Глава 3

Захтеви и ограничења алгоритама за проналажење најкраћег пута у видео играма

У претходном поглављу разматрали смо начине претварања тродимензионих и дводимензионих окружења у навигационе графове. У овом поглављу бавићемо се захтевима и проблемима приликом претраге претходно креираних навигационих графова.

3.1 Проблем претраге навигационог графа

Један од основних захтева у видео игри јесте обезбедити карактеру из игре механизам за успешно кретање кроз окружење игре на захтевани начин. Уколико карактер није у могућности да се успешно креће кроз своје окружење, то може довести до великог броја навигационих грешака.

Навигационе грешке могу настати услед постојања опционих путања, грешака приликом избегавања препрека, или чак у самом проналажењу најкраћег пута. Грешке овог типа су знатно видљивије из перспективе играча него било које друге грешке (чак и веће грешке) у вештачкој интелигенцији карактера. Играчи чак имају већу толеранцију код већих грешака у вештачкој интелигенцији него код грешака на нижен нивоу, најчешће зато што играч и

не примети да је грешка направљена. Нпр. играч вероватно неће ни приметити да постоји непрецизност приликом испаљивања метака из пушке, али ће увек приметити прелазак карактера из једне у другу собу директно кроз зид.

Кретање карактера у видео игри састоји се како од покрета у одређеном смеру тако и од планирања самог кретања. Физичко кретање карактера изводи се коришћењем скретања, кретања у јату (енг. flocking) [3], или неким физички заснованим системом. Ови системи кретања постепено приближавају карактер све ближе циљној локацији сваким протеклим фрејмом. Као такви ови системи имају проблем уколико циљна локација није директно достижна од тренутне локације карактера.

Окружења видео игара годинама расту како по величини тако и по њиховој комплексности. Како би се карактери неометано и исправно кретали по окружењу видео игре, систем кретања видео игре мора бити подржан системом планирања који ће гарантовати да карактер увек стигне на циљну локацију, уколико је та локација заиста и достижна.

Систем планирања кретања карактера одговоран је за проналажење листе позиција у окружењу које воде до циљне локације. Свака позиција у листи мора бити директно достижна из претходне позиције у листи. Ова листа формира путању кроз окружење видео игре која, ако је испраћена од стране карактера, води карактер до крајњег циља.

Само праћење пронађене путање изводи се од стране механизма за кретање карактера и гарантовано успева јер је раније систем за планирање обезбедио да свака позиција у путањи буде директно достижна из позиције која њој претходи. У данашњим видео играма најчешће постоји велики број различитих карактера који се крећу кроз окружење видео игре. Међу њима може доћи до колизија (судара) уколико им се путање кретања на неким местима преклапају, тако да системи за кретање карактера често имају подмеханизам за избегавање колизија.

У основи, проналажење путање кроз окружење је проблем претраге и као такав може бити решен коришћењем неког алгоритма за претрагу. Како навигациони граф садржи сва поља које је могуће обићи у окружењу игре он представља простор претраге за проблем проналажења најкраћег пута. Претрага ових навигационих графова захтева коришћење алгоритама за претрагу

графова што редукује проблем проналажења најкраћег пута на ништа друго до проблем претраге графа.

Решење проблема проналажења најкраћег пута је путања, која је дефинисана као листа тачака, ћелија или чворова кроз које карактер у игри мора проћи како би од почетне локације дошао до циљне локације. Као и за сваки проблем постоје различити захтеви и ограничења која су пред њим постављени.

3.2 Захтеви алгоритама за проналажење најкраћег пута

Посматрано на основном нивоу, постоје два фундаментална захтева које алгоритами за проналажење најкраћег пута требају испунити:

- Комплетност алгоритма: односи се на могућност алгоритма да пронађе решење у оквирима простора претраге. Алгоритам се сматра комплетним уколико он гарантује проналажење решења уколико решење заиста и постоји. Супротно, алгоритам који не гарантује проналажење решења сматра се некомплетним алгоритмом.
- Оптималност алгоритма: односи се на квалитет решења које алгоритам претраге пронађе. Уколико се гарантује да алгоритам проналази најбоље (оптимално) решење, алгоритам се тада сматра оптималним алгоритмом. Алгоритми који враћају резултате који су приближни оптималном решењу, али не и оптимални називамо апроксимативним или скоро оптималним алгоритмима.

За сваки проблем проналажења најкраћег пута може постојати више различитих решења од којих нека решења могу бити боља од других. Како би се квантификаовао квалитет неког решења и омогућило упоређивање једног решења са другим, користи се метрика оптималног решења. Ова метрика оптималне путање се користи при квантификовању квалитета алгоритма претраге обезбеђујући средство за мерење просечног квалитета решења пронађених неким алгоритмом.

Као што је речено алгоритми за проналажење најкраћег пута имају простор за претрагу који је представљен навигационим графом. Могуће је да имамо

висок ниво апстракције између оригиналног окружења и репрезентације навигационим графом. У таквим случајевима, најбоља пронађена путања од стране алгоритма за претрагу може естетски изгледати веома лоше када се употреби од стране карактера у оригиналном окружењу игре. Како би се квалитет финалне путање побољшао, могуће је извршити корак заобљавања путање (енг. path smooting).

3.2.1 Оптималност путање

Метрика оптималне путење нам говори колико је неко решење близу доступног оптималног решења. То значи, да би било могуће мерити квалитет пронађеног решења, оптимално решење мора унапред бити познато. Приликом процене оптималности неког неоптималног алгоритма, потребан је и оптималан алгоритам ради обезбеђивања контролног решења.

Решења се затим међусобно пореде користећи одређене особине добијених решења. Нпр. када је у питању проналажење најкраћег пута у видео играма, оптималност путање је најчешће одређена на основу дужине путање. Проценат оптималности коначне предложене путање представљен је процентом оптималности датог решења у поређењу са оптималним решењем. Проценат оптималности се рачуна на следећи начин:

$$Procenat\ optimalnosti = \frac{S_{predlozeno} - S_{optimalno}}{S_{optimalno}}$$

где је $S_{predlozeno}$ вредност предложеног решења, а $S_{optimalno}$ вредност оптималног решења. Оптимално решење је увек пожељно, али се у неким случајевима мора направити компромис ради повећања перформанси алгоритма.

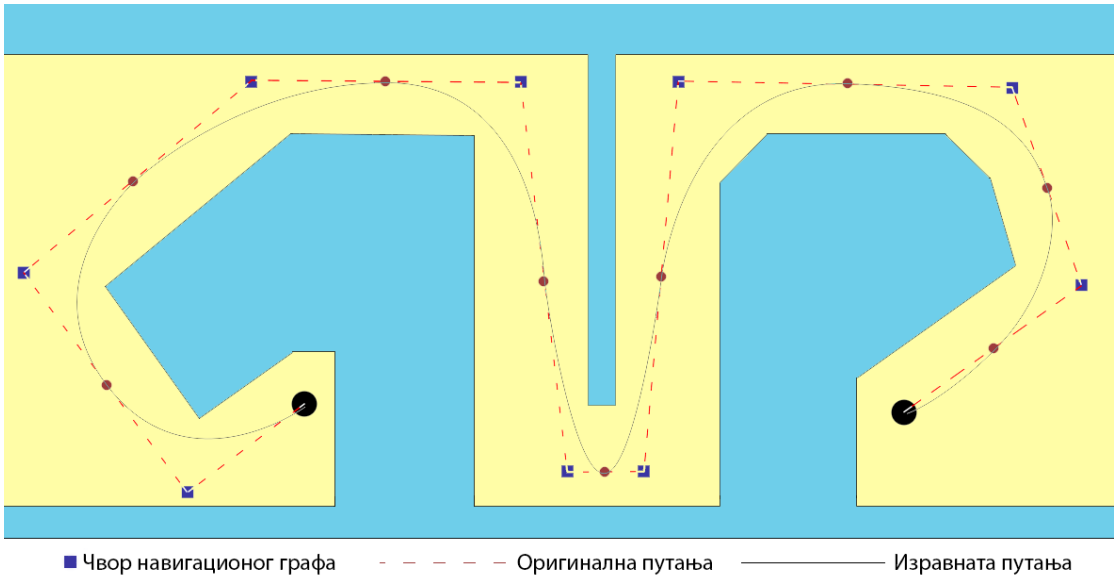
3.2.2 Заобљавање путање

Путања која је добијена алгоритмом за проналажење најкраћег пута се састоји од листе чворова из навигационог графа које је редом потребно обићи да би се дошло до циљног одредишта. Зависно од тога на који начин је представљен навигациони граф, може се догодити да навигациони граф не репрезентује сасвим детаљно окружење из видео игре. Овакав случај најчешће имамо код

навигационог графа заснованог на маркерима где сваки чвор навигационог графа представља велику област у окружењу видео игре.

Услед оваквог високог нивоа апстракције насталог коришћењем навигационих графова, праћење путање добијене алгоритмом за проналажење најкраћег пута може довести до проблема као што су заглављивање карактера, добијање подоптималне путање иако је пронађен оптимални навигациони граф.

Слика 3.1 приказује оптималну путању у пронађеном навигационом графу, која је очигледно подоптимална у односу на финално решење. Праћење подоптималне путање добијене помоћу навигационог графа такође може довести до естетски лошег начина кретања карактера у видео игри које не би изгледало реалистично.



Слика 3.1: Ефекат заобљавања путање резултујуће путање добијене преко навигационог графа

Како би се та реалистичност кретања карактера повећала, често је накнадно након проналаска путање коришћењем алгоритма за проналажење најкраће путање, потребно применити корак заобљавања на резултујућу путању.

Постоји неколико техника за побољшање квалитета резултујуће путање. Нпр. једноставно спајање чворова резултујуће путање кривом ће на добар начин повећати реалистичност и ниво заобљавања.

Више о техникама заобљавања путање може се наћи у [4],[5] и [6].

Понекад корак заобљавања путање није ни потребан зависно од начина на који је реализовано кретање карактера у видео игри. Нпр. могуће је употребити технику побољшаног скретања за кретање између чворова у путањи, у том случају сама техника скретања би аутоматски заоблила путању.

3.3 Ограничења алгоритама за проналажење најкраћег пута

Ограничења алгоритама за проналажење најкраћег пута можемо поделити у две категорије:

- Временска ограничења перформанси: односе се на временску цену процесирања претраге приликом решавања постављеног проблема, као и на време које је потребно за извршавање претраге.
- Меморијска ограничења: односе се на количину меморије која је потребна приликом решавања постављеног проблема, као и од меморијских тршкова саме инстанце алгоритма претраге.

Каква ће бити ограничења алгоритама за проналажење најкраћег пута највише зависи од типа видео игре и од платформе на којој ће игра бити покретана. Стратегије у реалном времену (енг. RTS - real time strategy) представљају посебно велики изазов за проблем проналажења најкраћег пута. RTS игре су засноване на приступу одозго према доле (енг. top-down) и могу садржати и на хиљаде карактера на мапи којима је неопходан неки независни или полу независни вид кретања.

Сложене акције кретања карактера захтевају употребу акција планирања путање. Како на мапи постоји велики број карактера са високим нивоом независности у кретању, систем за проналажење најкрачег пута често може имати задатак проналажења путање за велики број карактера истовремено. Планирање путање за огроман број карактера истовремено представља озбиљан изазов и у неким случајевима одузима чак половину процесорског времена видео игре.

Чак и када имамо тип игре који има ниже захтеве, акције планирања путање одузимају велику количину процесорског времена видео игре.

Када је реч о ограничењима алгоритама за проналажење најкраћег пута која зависе од платформе на којој се игра извршава, ограничења се односе на просецорску снагу и количину меморије коју платформа поседује.

3.3.1 Временска ограничења перформанси

Најчешћа мера која се користи за процену перформанси неког алгоритма је мера комплексности израчунавања. Комплексност израчунавања односи се на цену извршавања алгоритма (временску и просторну) у зависности од броја операција које је потребно извршити како би се алгоритам успешно извршио. Комплексност израчунавања најчешће представљамо помоћу O-нотације.

Време извршавања алгоритма може бити процењено или измерено за неке конкретне улазне вредности и неко конкретно извршавање. Но, време извршавања програма може бити описано општије, у виду функције која зависи од улазних аргумената.

Често се алгоритми не извршавају исто за све улазе истих величина, па је потребно наћи начин за описивање и поређење ефикасности различитих алгоритама. Анализа најгорег случаја заснива процену сложености алгоритма на најгорем случају (на случају за који се алгоритам најдуже извршава — у анализи временске сложености, или на случају за који алгоритам користи највише меморије — у анализи просторне сложености).

Нажалост, како O-нотација представља процену перформанси алгоритма на основу најгорег случаја то може довести до погресног тумачења перформанси алгоритма.

Услед ограничених могућности коришћења комплексности израчунавања, време извршења алгоритма или време претраге у случају алгоритама за проналажење најкраћег пута се најчешће користи за поређење алгоритама претраге када је у питању време израчунавања.

Постоје две најчешће коришћене методе евалуације алгоритама претраге на основу времена претраге које се користе за њихово поређење:

- Укупно време претраге: представља укупно време потребно алгоритму претраге да пронађе резултујућу путању на фиксираном скупу проблема током једне претраге.

- Просечно време претраге: представља просечно време потребно алгоритму претраге да пронађе резултујућу путању на фиксираном скупу проблема током неколико пута извршене претраге.

Нажалост, постоји проблем и код коришћења времена претраге као метрике. Време извршења алгортима зависи како од имплементације алгоритама тако и од платформе на којој је алгоритам покренут, што говори да није добро користити само време претраге као метрику временске перформансе алгоритма.

Временске перформансе алгоритма претраге директно зависе и од броја претражених чворова (испуњености простора претраге) у току претраге. Додатно, поред броја претражених чворова у току претраге алгоритми претраге често могу поново посетити неке већ претражене чворове, што додатно увећава време потребно за проналажење коначне путање.

Број претражених чворова и претрага већ посећених чворова имају различите цене перформанси, претрага већ посећених чворова је најчешће мањег утицаја на временсе перформансе алгоритма од броја претражених чворова.

Коришћење броја претражених чворова и броја посећених чворова као метрике је препоручљиво јер су ове метрике независне од импелемнтације алгоритма и од платформе на којој се алгоритам извршава.

И метрика времена претраге и метрика броја претражених чворова и претраге већ посећених чворова имају своја ограничења када се користе појединачно. Из тог разлога препоручљиво је да се поређење временских перформанси алгоритама врши коришћењем обе наведене метрике комбиновано.

3.3.2 Меморијска ограничења

Меморијско ограничење алгоритама за проналажење најкраћег пута углавном зависи од платформе на којој се игра извршава. Нпр. алоцирање 25MB

простора за податке потребне приликом проналажења најкраћег пута на рачунару је сасвим прихватљиво, док се за податке потребне приликом проналажења најкраћег пута на конзоли за видео игре алоцира између 1MB и 2MB.

Оваква ограничења у количини меморије која се алоцира за податке приликом коришћења алгоритама за проналажење најкраће путање а која зависе од платформе утичу на одабир алгоритма претраге који се користи приликом планирања путење.

Укупна меморијска цена алгоритма претраге може се поделити у две категорије:

- Цена података по чвору: односи се на количину података која настаје приликом сваког проласка кроз неки чвор мреже. Овако настали подаци се касније користе приликом проналажења најкраћих путања као и приликом конструисања коначног решења.
- Цена података по алгоритму: различитим типовима алгоритама претраге потребна је различита количина меморије за податке који се користе приликом претраге како би се унапредиле перформансе или квалитет алгоритма претраге. Нпр. код хиерархијског A* алгоритма (енг. HPA* - Hierarchical Path-Finding A*) потребна је одређена количина меморије за складиштење апстракција коришћених хиерархија [7].

Цена података по алгоритму, уколико имамо статичко окружење у игри, остаје константна, док промене у окружењу када имамо игру са динамичким окружењем могу значајно утицати на цену података по алгоритму. У већини случајева, укупна цена података по чвору је доста већа од цене података по алгоритму. То је и разлог да приликом оптимизације меморије алгоритма за претрагу главни фокус буде на редуковању цене података по чвору.

Како се простор за податке по чвору алоцира приликом првог проласка кроз неки од чворова током претраге, што је већи простор претраге алгоритма, то ће бити потребна и већа количина меморије за складиштење тих података. Ако се неки чвор посети више пута то не утиче на повећање меморије која је потребна алгоритму претраге. Цена података по чвору алгоритма претраге такође се назива и просторном комплексношћу алгоритма.

Укупан меморијски простор потребан алгоритму је тешко одредити, јер постоји велики број могућих начина претраге простора за неки проблем претраге, што зависи од проблема претраге који се решава.

За сваки алгоритам могуће је одредити цену меморијског простора у најгорем случају, али поред рачунске комплексности таквог израчунавања доводи се у питање и колико је такав податак уопште употребљив. Како би се успешно упоредила цена меморијског простора два алгоритма, потребно је израчунати просечну и највећу могућу цену меморијског простора оба алгоритма. Како цена меморијског простора зависи од величине простора који се претражује, упоређивање просторних комплексности засновано је на величини простора претраге сваког од алгоритама.

У скорије време доста се радило на креирању алгоритама претраге са редукованом потребном меморијом или са ограничавањем меморије алгоритму претраге [8]. На том пољу је учињен и значајан напредак када је у питању смањење просторне комплексности алгоритама претраге. Када су у питању алгоритмими претраге са редукованом меморијом, редуковање цене често може имати негативан ефекат како на оптималност добијеног решења тако и на време потребно алгоритму претраге за проналажење решења.

3.4 Проналажење најкраћег пута у динамичком окружењу

Готово све данашње сложеније видео игре се одигравају у динамичким окружењима. То значи да се окружење видео игре у току извођења игре може више пута променити услед различитих фактора (експлозија, временске неприлике, физичко померање елемената у игри). Постоји висок ниво интеракције између играча и окружења што омогућава играчу да значајно измени изглед окружења игре својим акцијама. Пример такве измене окружења игре од стране играча у видео игри Company of Heroes приказан је на слици 3.2.

Неке игре се чак искључиво ослањају на динамичко окружење као један од главних аспеката у механици извођења игре (енг. gameplay). Пример таквих игара су серијали Company of Heroes, Worms, Battlefield.

Слика 3.2: Пример измене изгледа окружења игре у видео игри Company of Heroes. Почетно стање окружења игре (лево) и исто окружење након битке (десно)

У најновијој игри из Battlefield серијала Battlefield Hardline могуће је на једној од мапа уништити грађевинску дизалицу која приликом пада на земљу потпуно мења окружење игре уништивши околне зграде.

Услед различитих акција и услед високог нивоа деструктивности које данашње игре пружају играчима, окружење игре може се изменити и до непрепознатљивости. Области по којима је раније било могуће проћи могу постати непролазне услед новонасталих препрека и обрнуто. Овакве измене простора повећавају комплексност проблему проналажења најкраћег пута.

Како би се алгоритми за проналажење најкраћег пута могли носити са променом окружења игре, први приоритет јесте ажурирање навигационог графа. Сложеност ажурирања навигационог графа варира у зависности од начина представљања окружења навигационим графом. Начин представљања навигационим графом се зато мора пажљиво одабрати приликом његове употребе за проналажење најкраћег пута у динамичном окружењу.

Приликом кретања карактера кроз динамично окружење, проналажење најкраћег пута се обавља као и у статичним окружењима са додатком разматрања да у сваком моменту промена у окружењу игре може утицати на све тренутно испланиране путање.

Утицаји промене у окружењу на већиспланирану путању могу бити:

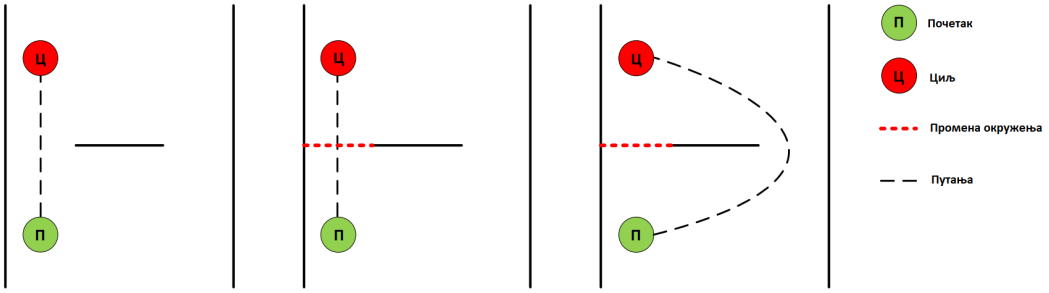
- непролазна путања: неки део претходно испланиране путање може постати непролазан услед промене окружења, видети слику 3.3.

- скраћење путање: може доћи до настанка пречице, ако услед промене окружења нови део мапе постане пролазан, тако да се претходно испланирана путања може додатно скратити, видети слику 3.4.

У оба случаја, планирање путање мора се извршити изнова макар у делу мапе који је измењен.

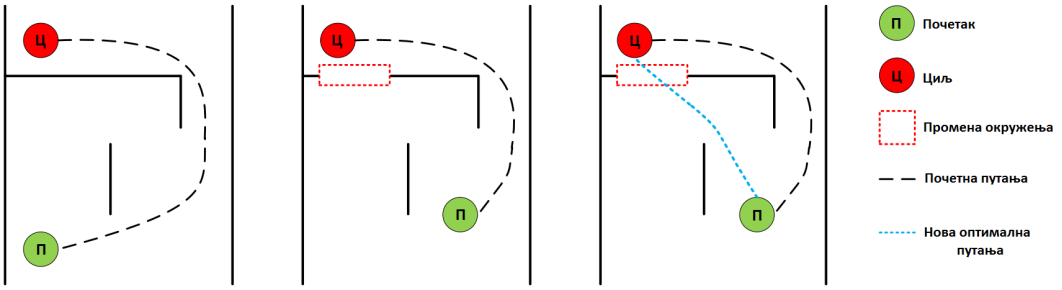
Уколико дође до прекидања испланиране путање услед промене окружења, откривање оваквог случаја је прилично једноставно. Сваку испланирану путању у близини настале промене окружења потребно је проверити како би се утврдило да ли је дошло до настанка неисправних чворова, тј. чворова који су недостижни након измене окружења (било да су постали непролазни или више не постоје).

Ако испланирана путања обухвата неисправан чвор, потребно је поново извршити планирање путање. Поновно планирање путање врши се новом претрагом најкраће путање са истим циљним чвором, али од тренутне позиције карактера у навигационом графу у моменту настанка промене у окружењу. Ова нова претрага путање у току кретања а услед насталих измена у окружењу се назива и акцијом репланирања путање.



Слика 3.3: Репланирање путање услед промене у окружењу. Почетна путања (лево), прекид почетне путање (средина) и репланирана путања (десно)

Откривање постојања нове оптималне путање за дати проблем претраге након настанка промене у окружењу игре је значајно сложенији задатак од обичне провере исправности већ постојеће путање. Свака потенцијална промена у окружењу игре, која доводи до отварања нових (раније непролазних) локација на мапи, може произвести ново оптимално решење раније решеног проблема претраге.



Слика 3.4: Репланирање путање услед промене у окружењу. Почетна путања (лево), настанак пречице (средина) и нова оптимална путања (десно)

На слици 3.4 је приказана таква ситуација када промена окружења у близини испланиране путање производи ново оптимално решење. Једини начин провере постојања нове оптималне путање јесте репланирањем путање у моменту настанка промене у окружењу и упоређивањем нове добијене путање репланирањем и већ постојеће путање. Ако је при репланирању коришћен оптималан (или скоро оптималан) алгоритам тада је поређење нове и постојеће путање редундантно, с обзиром да ће нова путања тада увек бити иста или боља од постојеће путање.

Провера постојања пречице увек захтева извођење репланирања путање приликом промене у окружењу, провера исправности путање тиме такође постаје редундантна јер ће путања свакако бити репланирана. Заправо више није потребно изводити било какве провере пошто се путања свих карактера у игри увек репланира приликом настанка промене у окружењу. Овакав систем гарантује да ће путање карактера у игри увек имати оптимално решење које се може пронаћи коришћеним алгоритмом претраге.

Свака промена у окружењу, при оваквој поставци ствари, резултовала би репланирењем свих претходно испланираних путања. У играма у којима имамо велики броја различитих карактера на мапи, једна промена у окружењу покренула би велики број захтева за репланирањем путање. Како је свака акција планирања путање атомична и како време извођења акције планирања путање одређује колико се акција планирања може покренути по фрејму, неопходно је користити веома ефикасан алгоритам претраге у играма са веома динамичним окружењем и великим бројем карактера.

Нажалост, чак и са веома ефикасним алгоритмом претраге дешава се да није могуће извршити акције репланирања у захтеваном временском року и време

одзива (енг. response time) може бити дуже од потребног.

3.5 Одабир одговарајућег алгоритма претраге

Од тога колико је неки алгоритам погодан за дато окружење и од његове ефикасности зависи и ефикасност читавог система за проналажење најкраћег пута. Савршени алгоритам претраге би требало да обезбеди оптималне путање без утрошка времена и меморије. Нажалост, како у пракси овако савршен алгоритам не постоји, потребно је пронаћи компромис с обзиром на значај важности оптималности путање и утрошка времена и меморије.

Који ће алгоритам претраге бити употребљен у некој видео игри зависи од следећих фактора:

- платформе којој је игра намењена,
- потребног броја карактера у игри,
- динамичности окружења игре.

Меморијско ограничење постављено алгоритму услед зависности од количине меморије платформе на којој се игра извршава веома је важан фактор приликом одабира алгоритма претраге који ће бити употребљен. У оваквом случају, најчешће је потребно направити велики компромис када је у питању брзина претраге, што опет зависи од окружења игре.

Разматрајући најгори могући случај, када је у питању утрошак меморије алгоритма, приликом његовог одабира обезбеђује се да алгоритам никада не прекорачи количину меморије која му је доступна на платформи.

Узимање у обзир само најгорег случаја, када је у питању меморијска цена алгоритма, није добра пракса зато што је у већини случајева потребно значајно мање меморије при решавању проблема претраге него што је то процењено у најгорем случају. Уколико је главни фактор приликом одабира алгоритма само најгори случај када је у питању меморијска цена алгоритма то доводи до превеликог редуковања меморијске цене алгоритма науштрб бољих перформанси алгоритма. Зато је боље, као факторе приликом одабира одговарајућег

алгоритма за претрагу када постоји меморијско ограничење, узети највишу и просечну меморијску цену алгоритма претраге.

Највишу и просечну меморијску цену алгоритма претраге потребно је одредити емпиријски кроз различите тестове у тест окружењу које представља скуп окружења у финалној игри. При коришћењу највише меморијске цене при одабиру алгоритма треба бити веома опрезан. Уколико је највиша меморијска цена алгоритма претраге близу меморијског лимита платформе, већа је шанса прекорачења меморијског лимита платформе него у случају када је та цена знатно нижа од лимита платформе.

Уколико алгоритам претраге прекорачи расположиву количину меморије може доћи до грешке у претрази или, код лоше имплементације алгоритма, пада апликације. Алгоритам који има нижу највишу меморијску цену од другог алгоритма који има нижу просечну меморијску цену бољи је кандидат за употребу у случају када постоји меморијско ограничење платформе.

Када говоримо о перформансама алгоритма, јасно је да што је алгоритам бржи, то је и бољи. Ипак одабир алгоритма првенствено зависи од меморијских ограничења. То значи да ће на крају бити одабран алгоритам који нуди најкраће време извршавања поштујући при томе меморијска ограничења платформе.

Глава 4

Најчешће коришћени алгоритми за проналажење најкраћег пута у видео играма

Проблем проналажења најкраћег пута у видео играма се у основи своди на проблем претраге графа, који се решава коришћењем алгоритама за претрагу графа. Постоји велики број алгоритама за претрагу графа али нису сви од њих погодни за коришћење у проналажењу најкраће путање у видео играма због ограничења перформанси и меморијског ограничења платформе за игру.

Алгоритме који се користе за проналажење најкраћег пута у видео играма можемо поделити у две основне групе: дискретне и континуалне алгоритме претраге.

Дискретни алгоритми претраге узимају у обзир изглед мапе (навигационог графа) у једном тренутку времена и нису погодни за коришћење у динамичном окружењу видео игре. Најчешће се користе уколико нема промена изгледа мапе током целог извођења игре.

У динамичним окружењима видео игре користе се континуални алгоритми који имају способност надградње тренутног изгледа навигационог графа на основу прикупљених података.

Размотримо ситуацију када карактер у игри има само основно знање о изгледу окружења игре (нпр. познаје само део мапе). Приликом кретања, карактер

подразумева да је све непознате чворове могуће обићи и планира путању на основу те претпоставке. Како карактер пролази кроз окружење, добија нове информације о окружењу којих раније није био свестан. Одређени чворови које је раније сматрао прелазним могу се испоставити непрелазним па путања карактера мора бити коригована након узимања овог новог податка о окружењу у обзир.

Када карактер наиђе на овакву промену у окружењу његова путања се репланира и карактер наставља кретање новом путањом. Кретање караткера кроз овакво окружење може изазвати велики број акција репланирања услед континуираних измена навигационог графа, свака акција репланирања покреће неки дискретни алгоритам претраге.

Цена покретања алгоритма дискретне претраге сваки пут приликом репланирања може бити превисока и може изазвати паузирање карактера на њиховој путањи кад год се догоди промена у окружењу. Потребан је алгоритам који врши јефтине локализоване измене путање по потреби у току кретања карактера. У овим случајевима користимо континуални алгоритам претраге.

Овде уочавамо разлику између дискретних и континуалних алгоритама претраге. Коришћењем дискретног алгоритма претраге за решавање оваквог проблема, добијамо решење након одређеног времена. У току тог времена дискретан алгоритам претраге не детектује промене у окружењу и не коригује пронађено решење ако је то потребно. Супротно, континуални алгоритам освежава изглед постојећег проблема током свог извршавања и поправља решење приликом промене у окружењу у току свог извршавања.

4.1 Дискретни алгоритми претраге

Дискретни алгоритми претраге се чешће користе у случају када нема промена изгледа мапе током целог извођења игре, мада се могу користити и у динамичком окружењу иако су за то погоднији континуални алгоритми претраге. Као два најчешће коришћена дискретна алгоритма претраге издвајају се Дијкстрин алгоритам и алгоритам A*.

4.1.1 Дијкстрин алгоритам

Едсгер Дијсктра представио је 1959 године нови алгоритам претраге који је по њему добио име Дијкстрин алгоритам. Дијкстрин алгоритам није оригинално дизајниран на начин на који би могао бити лако коришћен у видео играма. Дизајниран је првенствено за решавање проблема проналажења најкраћег пута у математичкој теорији графова.

Овај алгоритам се користи и данас, а примену је нашао и у проналажењу најкраћег пута у видео играма. Дијкстрин алгоритам представља метод проналажења најкраћег пута од почетног чвора до свих осталих чворова у тежинском графу. У таквом графу, свака тежина представља цену проласка преко неке од грана између два чвора.

Дијкстрин алгоритам врши претрагу сваког чвора у задатом графу, при том се током самог процеса претраге за сваки чвор чувају подаци о најкраћој путањи од почетног чвора до њих. Након што се алгоритам изврши до краја, путања се затим реконструише од циљног до почетног чвора користећи податке о најкраћој путањи смештене у сваком од чворова. Ово је итеративан алгоритам и сваком итерацијом претрага се врши за појединачан чвор.

Решење оваквог проблема обезбеђује проналажење најкраћег пута који је и тражен (пронађени су накраћи путеви до свих чворова од почетног чвора, па тако и до циљног чвора), али код Дијкстриног алгоритма постоји и непотребно проналажење најкраћих путева до свих осталих чворова који се након претраге заправо одбаце и непотребни су.

Због ових проблема, Дијкстрин алгоритам се данас веома ретко користи као главни алгоритам за проналажење најкраћег пута, већ се више користи за неке анализе нивоа и мапа. Важан је као алгоритам који се користи за тактичку анализу, као и у многим другим областима вештачке интелигенције у видео играма.

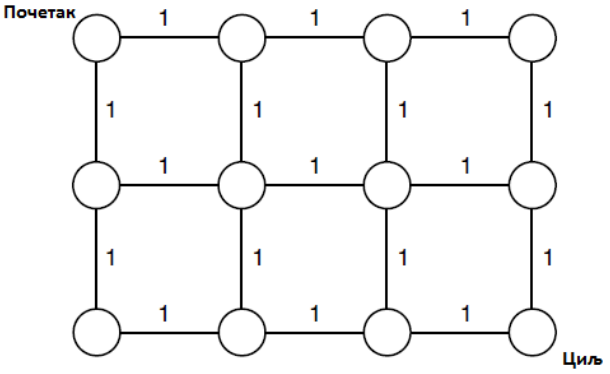
4.1.1.1 Поставка проблема

За дати граф (усмерени тежински граф са ненегативним тежинама) и два чвора (почетни и циљни) у том графу, потребно је пронаћи путању са минималном ценом међу свим могућим путањама од почетног до циљног чвора.

Могуће је да постоји и већи број путања са истом минималном ценом . Нпр. на слици 4.1 је приказано 10 различитих путања са истом минималном ценом. Када имамо овакав случај, одабира се једна од путања без обзира која од њих је у питању.

Путања коју очекујемо добити као решење састоји се од низа грана. Два чвора могу бити повезана са више грана од којих свака може имати различите цене. Алгоритам за проналажење најкраћег пута узима у обзир само грану са најмањом ценом међу гранама између два иста чвора.

Међутим у видео играма може доћи до измена цена грана током трајања игре или уколико имамо карактере у игри са различитим начинима кретања (нпр, неки се брже крећу кроз воду, неки не) па је стално праћење свих грана између два чвора ипак корисно.



Слика 4.1: Више најкраћих путања са истом ценом између почетног и циљног чвора

4.1.1.2 Решење проблема

Дијкстрин алгоритам има карактеристику ширења претраге од почетног чвора дуж грана ка осталим чворовима. Ширењем претраге ка удаљеним чворовима, током проласка кроз граф чува се информација о путањи која је пређена до тих удаљених чворова. Коначно доласком до циљног чвора могуће је реконструисати најкраћу путању до почетног чвора користећи информације о пређеној путањи прикупљене током претраге.

Детаљније, Дијкстрин алгоритам се састоји из итерација. Сваком итерацијом разматра се један чвор графа и његове гране ка осталим чворовима. У првој итерацији разматра се почетни чвор. У свакој наредној итерацији одабира

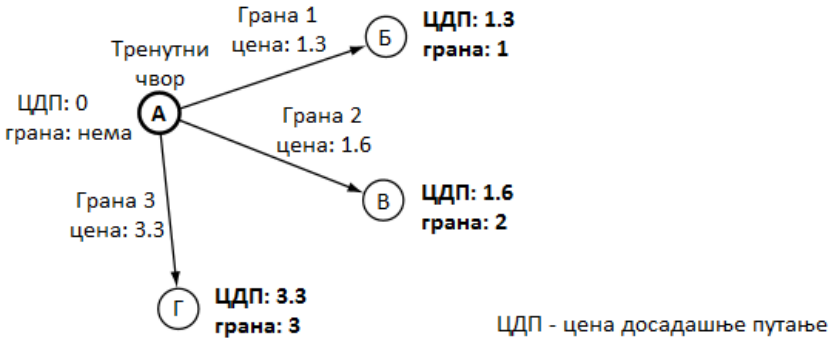
се следећи чвор на основу алгоритма. Чвор који се разматра у итерацији је тренутни чвор.

Обрада тренутног чвора

Током итерације, Дијкстрин алгоритам разматра сваку грану ка даљим чворовима тренутног чвора. За сваку грану проналази се чвор који та грана повезује са тренутним чвором и чува се информација о тренутној збирној цени од почетног чвора (цена досадашње путање), као и грана од које се дошло до тог чвора.

У првој итерацији, када је почетни чвор заправо тренутни чвор, цена досадашње путање сваког чвора на крају гране која их спаја са почетним чвором је заправо цена саме гране.

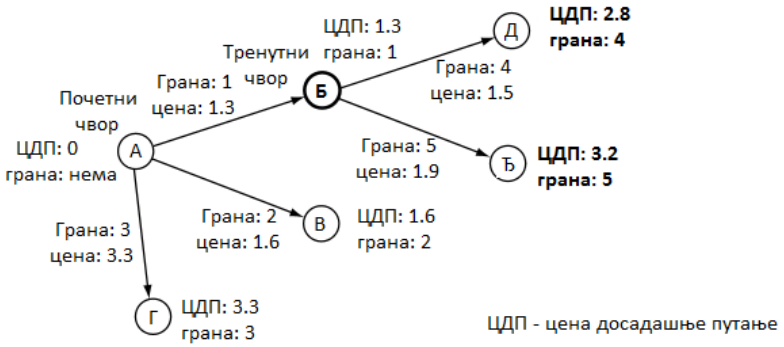
На слици 4.2 је приказана ситуација након прве итерације. Сваки чвор који је повезан са почетним чвором има цену досадашње путање једнаку цени гране која је до њега водила, као и информацију о грани којом је чвор повезан са претходним (у овом случају почетним) чвором.



Слика 4.2: Дијкстрин алгоритам у првој итерацији где је почетни чвор уједино и тренутни чвор

За све наредне итерације, цена досадашње путање за крајње чворове у тој итерацији једнака је суми цене гране која води до крајњег чвора у тој итерацији и цене досадашње путање тренутног чвора (чвора с којим је граном повезан крајњи чвор у тој итерацији).

На слици 4.3 је приказана следећа итерација истог графа приказаног на слици 4.2. У овом случају је цена досадашње путање која је смештена у чвору Д једнака суми цене досадашње путање чвора Б и тежине (цене) гране 4 која повезује чвор Б са чвором Д.



Слика 4.3: Дијкстрин алгоритам у другој итерацији

У имплементацији алгоритма нема разлике у првој и наредним итерацијама. Постављањем цене досадашње путање почетног чвора на 0 (пошто почетни чвор нема удаљеност од самог себе) исти код се може користити код сваке итерације.

Листа отворених чворова и листа затворених чворова

У Дијкстрином алгоритму користе се две различите листе чворова током његовог извршавања. То су листа отворених чворова и листа затворених чворова. У листи отворених чворова се чувају сви чворови за које алгоритам зна, али их још није обрадио кроз неку од досадашњих итерација. У листи затворених чворова се чувају сви чворови које је алгоритам до датог тренутка обрадио у некој од итерација.

У почетној итерацији листа отворених чворова садржи само почетни чвор (са ценом досадашње путање 0), а листа затворених чворова је празна.

Сваки чвор може бити у једној од три категорије:

- Може бити у листи затворених чворова, након што је обрађен у некој од итерација алгоритма.
- Може бити у листи отворених чворова, уколико је посећен у итерацији у којој је неки други чвор посматран као тренутни а још увек није био тренутни чвор у некој од итерација.
- Чвор није ни у листи отворених чворова ни у листи затворених чворова. Такве чворове називамо непосећеним чворовима.

У свакој итерацији алгоритам одабира чвор из листе отворених чворова који има најнижу цену досадашње путање и поставља га за тренутни чвор. Након извршене итерације обрађени чвор се уклања из листе отворених чворова и смешта се у листу затворених чворова.

Због концепта ширења претраге најчешће је тренутни чвор гранама повезан са непосећеним чворовима, али могуће је да се током итерације догоди да се разматрају гране од тренутног чвора до чворова који су већу листи отворених чворова или листи затворених чворова. У таквим случајевима потребно је мало другачије поступити са оваквим чворовима.

Рачунање цене досадашње путање већ отворених или затворених чворова

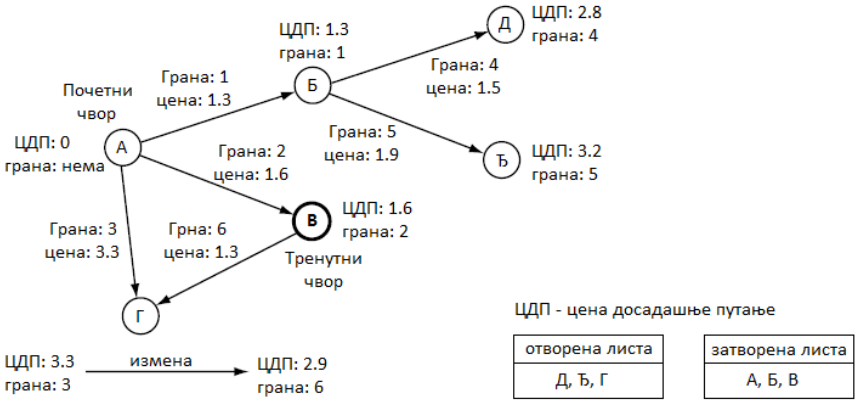
Уколико се током неке од итерација разматра већ отворен или затворен чвор, такав чвор већ има сачуване информације о цени досадашње путање и грани преко које се дошло до тог чвора. Ако у овом случају једноставно упишемо информацију о новој цени досадашње путање и нову грану до датог чвора може се уништити све што је алгоритам раније пронашао за дати чвор, а постоји шанса да је претходно пронађена путања била боља (са нижом ценом досадашње путање).

Уместо таквог приступа, проверава се да ли је нова путања до датог чвора боља од већ раније пронађене путање. Израчуна се цена досадашње путање за нову путању и уколико је она већа од већ сачуване информације о цени досадашње путање за дати чвор (у већини случајева нова цена ће заиста бити већа од раније сачуване), није потребно мењати претходне информације за дати чвор и није му потребно мењати листу у којој је тренутно.

Уколико је нова цена досадашње путање нижа од већ сачуване информације о цени досадашње путање за дати чвор, потребно је изменити информацију на нову вредност цене досадашње путање и сачувати нову грану којом се дошло до датог чвора. Чвор је затим потребно сметити у листу отворених чворова уколико већ није био у њој, а ако је чвор био у листи затворених чворова потребно га је и уклонити из листе затворених чворова.

На слици 4.4 приказан је случај када је потребно изменити цену досадашње путање и информацију о грани којом се дошло до чвора за чвор који је већ у

листи отворених чворова. Нова путања, преко чвора В до чвора Г, је боља од раније пронађене путање па је информацију у чвору Г потребно изменити.



Слика 4.4: Измена цене досадашње путање на отвореном чвору

Завршетак алгоритма

Дијкстрин алгоритам завршава рад када је листа отворених чворова празна. Размотрен је сваки чвор у графу до којег се дошло од почетног чвора, а сви ти чворови се налазе у листи затворених чворова.

Приликом проналажења најкраћег пута могуће је прекинути са извршавањем алгоритма чим се дође до циљног чвора. У том случају може се догодити да се даљим разматрањем преосталих отворених чворова пронађе боља путања. Извршавање алгоритма је потребно прекинути када је цена досадашње путање циљног чвора најнижа.

Ако поново размотримо слику 4.4, уколико је Г циљни чвор, први пут ће чвор Г бити пронађен током обраде чвора А. Уколико алгоритам ту заврши са радом, јер је дошао до циљног чвора Г, путања ће бити А-Г, што није најкраћа путања. Зато је извршавање Дијкстриног алгоритма потребно прекинути тек након што је цена досадашње путање циљног чвора најнижа.

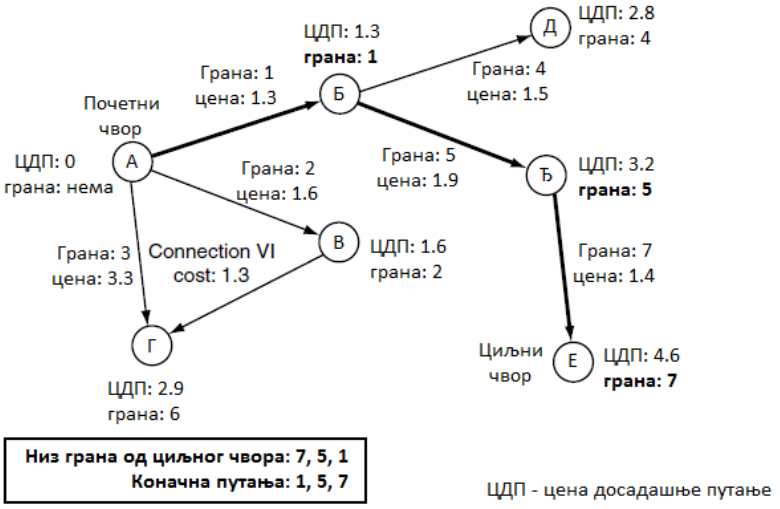
У пракси се ово правило често занемарује јер је најчешће прва пронађена путања до циљног чвора и најкраћа, па чак и уколико постоји краћа путања одабира се прва пронађена.

Реконструкција најкраће путање

Последњи корак Дијкстриног алгоритма је реконструкција најкраће путање. Креће се од циљног чвора и за њега се тражи информација преко које гране је алгоритам дошао до њега. Затим се процедура даље наставља прикупљајући информације о гранама којима се враћамо за сваки наредни чвор у путању све док се не дође до почетног чвора.

Добијена листа грана је исправна, али у погрешном редоследу, па је потребно обрнути листу и добија се коначна листа грана која чини најкраћу путању од почетног до циљног чвора.

На слици 4.5 приказано је реконструисање најкраће путање за дати алгоритам, ако је А почетни чвор а Е циљни чвор.



Слика 4.5: Реконструкција најкраће путање

4.1.1.3 Код Дијкстриног алгоритма

Дијкстрин алгоритам као аргументе узима навигациони граф, почетни и циљни чвор. Вредност коју Дијкстрин алгоритам враћа након извршења представља листу грана које сачињавају најкраћу путању од почетног до циљног чвора.

```
1 def pathfindDijkstra(graph, start, end):
2
3 # Struktura koja se koristi radi cuvanja
4 # porataka u o svakom od cvorova
5 struct NodeRecord:
6 node
7 connection
```

```
costSoFar

# Inicijalizacija podataka za pocetni cvor
startRecord = new NodeRecord()
startRecord.node = start
startRecord.connection = None
startRecord.costSoFar = 0

# Inicijalizacija otvorene i zatvorene liste cvorova
open = PathfindingList()
open += startRecord
closed = PathfindingList()


# Obrada cvorova kroz iteracije
while length(open) > 0:

# Pronaci element sa najnizom vrednoscu iz otvorene liste
# (posmatrajuci cenu dosadasnje putanje cvorova)
current = open.smallestElement()

# Ukoliko je trenutni cvor jednak ciljnom cvoru
# prekinuti izvr avanje algoritma.
if current.node == goal: break

# U suprotnom pronaci grane ka drugim cvorovima od trenutnog cvora.
connections = graph.getConnections(current)

# Proci kroz sve grane koje su pronadjene.
for connection in connections:

# Pokupiti procenu cene za ciljni cvor
endNode = connection.getToNode()

endNodeCost = current.costSoFar +
connection.getCost()

# Preskociti cvor ako je zatvoren
if closed.contains(endNode): continue

# .. ili ako je otvoren a pronadjena je
# losija putanja
else if open.contains(endNode):

```

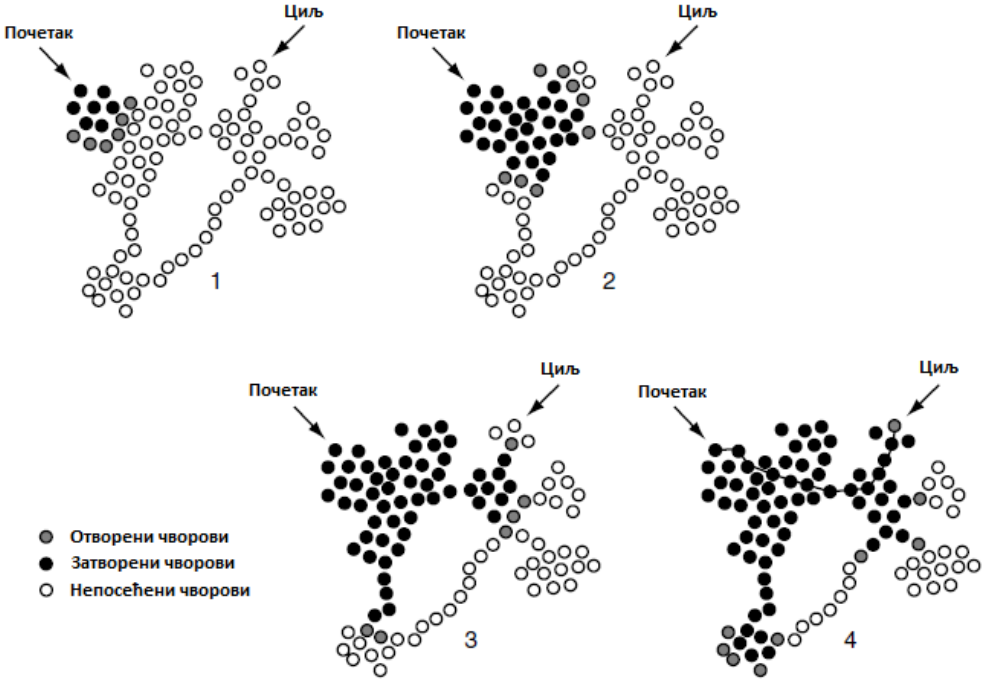
```
52 # Pronalazenje podataka u otvorenoj listi
53 # koji odgovaraju ciljnom cvoru.
54 endNodeRecord = open.find(endNode)
55
56 if endNodeRecord.cost <= endNodeCost:
57 continue
58
59 # U suprotnom imamo neposecen cvor
60 # pa se za njega cuvaju podaci
61 else:
62 endNodeRecord = new NodeRecord()
63 endNodeRecord.node = endNode
64
65 # Ukoliko je potrebno azurirati podatke
66 # Azurira se cena i grana
67 endNodeRecord.cost = endNodeCost
68 endNodeRecord.connection = connection
69
70 # Dodati cvor u otvorenu listu
71 if not open.contains(endNode):
72 open += endNodeRecord
73
74 # Nakon sto je zavrsen prolayak kroz grane
75 # trenutnog cvora, dodati cvor u zatvorenu listu
76 # i ukoloniti cvor iz zatvorene liste.
77 open -= current
78 closed += current
79
80 # Ukoliko trenutni cvor nije ciljni cvor
81 # ili kada nema vise cvorova za pretragu
82 if current.node != goal:
83
84 # Ponestalo je cvorova a resenje
85 #nije pronadjeno.
86 return None
87
88 else:
89 # Napraviti listu u koju je
90 # potrebno smestiti konacnu putanju
91 path = []
92
93 # Vracajuci se unazad formirati konacnu putanju,
94 # sakupljajuci grane u listu
95 while current.node != start:
```

```
96  path += current.connection
97  current = current.connection.getFromNode()
98
99  # Obrnuti listu kako bi sadzala grane
100 # konacne putanje sortirane od pocetnog
101 # ka ciljnom cvoru.
102 # Vratiti listu kao rezultat.
103 return reverse(path)
```

4.1.1.4 Слабости Дијкстриног алгоритма

Основни проблем код коришћења Дијкстриног алгоритма јесте што се њиме претражује читав граф у потрази за најкраћим путем, и тако се посети превелик број чворова које не би требало ни узети у разматрање чиме би се скратило време проналажења најкраће путање.

Ово је корисно уколико се тражи најкраћа путања до сваког чвора у графу (проблем за који је Дијкстрин алгоритам и дизајниран), али то ствара и превелик трошак у времену и меморији приликом проналажења најкраћег пута између два одређена чвора. На слици 4.6. је визуелно представљено колико непотребних чворова се Дијкстриним алгоритмом обиђе током типичног покретања алгортима (гране су уклоњене ради једноставности приказа).



Слика 4.6: Обилазак чворова у неколико корака

У фази 4 приказано је стање три листе у моменту прекида рада алгоритма. Линијом је проказана пронађена најкраћа путања између почетног и циљног чвора. Примећује се да је већи део мапе претражен, чак и делови који су јако далеко од пронађене најкраће путање.

Испуна Дијкстриног алгоритама представља број чворова који су разматрани али нису постали део резултујуће путање. Потребно је размотрити што мањи број чворова, јер се тако смањује потребно време за обраду чворова.

Понекад ће Дијкстрин алгоритам пронаћи најкраћу путању уз релативно малу испуну, али тај сучај је веома редак и представља изузетак. У већини случајева испуна је преко 70 процената од укупног броја чворова.

Алгоритми са високом вредношћу испуне, као Дијкстрин алгоритам, су неефикасни приликом проналажења најкраћег пута између тачно два одређена чвора, стога се такви алгоритми ретко и користе у оваквим случајевима. Најчешће коришћени дискретни алгоритам за проналажење најкраћег пута у видео играма јесте A* алгоритам. Алгоритам A* може се сматрати верзијом Дијкстриног алгоритма са минималном испуном.

4.1.2 Алгоритам A*

Синоним за проналажење најкраћег пута у видео играма јесте алгоритам A*. Овај алгоритам је веома популаран због своје једноставности имплементације, добре ефикасности, и веома великих могућности оптимизовања.

Већина система за проналажење најкраћег пута у видео играма који су се појавили у последњих 10 година користе неку од варијација алгоритма A* за своју основу, а његова примена иде и далеко даље од проналажења најкраћег пута у видео играма.

Дијкстрин алгоритам увек бира чвор из отворене листе с најнижом ценом досадашње путање, док хеуристика A* алгоритма бира чвор који ће највероватније бити део најкраће путање од почетног до циљног чвора. За разлику од Дијкстриног алгоритма, алгоритам A* је дизајниран за проналажење најкраћег пута између два конкретна чвора. Алгоритам A* увек враћа тачно једну путању од почетног до циљног чвора.[9]

Проналажење најкраћег пута алгоритмом A* у многоме зависи од одабране хеуристике. Уколико се користи лоша хеуристика проналажење најкраћег пута може трајати дуже него што је очекивано.

4.1.2.1 Поставка проблема

Поставка проблема алгоритма A* је идентична поставци проблема Дијкстриног алгоритма. За дати граф (усмерени, са ненегативним тежинама грана) и два чвора у том графу (почетни и циљни чвор), потребно је пронаћи путању са најнижом ценом међу свим могућим путањама између почетног и циљног чвора. Одабрати било коју путању са минималном ценом коштања уколико их има више, а као решење представити листу грана које повезују почетни и циљни чвор на тај начин.

4.1.2.2 Решење проблема

Уместо сталног одабира чвора са најнижом ценом досадашње путање из отворене листе као што је случај код Дијкстриног алгоритма, код алгоритма A* одабира се онај чвор који ће највероватније бити део најкраће путање. Одабир чвора који ће највероватније бити део коначне најкраће путање врши се помоћу хеуристике. Уколико је хеуристика добра (прецизна у процени одабира чвора) тада је и алгоритам A* ефикасан. У случају да је одабрана хеуристика јако лоша, перформансе алгоритма A* могу некада бити чак и лошије од Дијкстриног алгоритма.

Детаљније, алгоритам A* се састоји из итерација. Сваком итерацијом разматра се један чвор графа и његове гране ка осталим чворовима. Чвор који се разматра (тренутни чвор) у итерацији одабира се коришћењем алгоритма одабира који је сличан као код Дијкстриног алгоритма, са значајном разликом да је код алгоритма A* у одабир укључена и хеуристика.

Обрада тренутног чвора

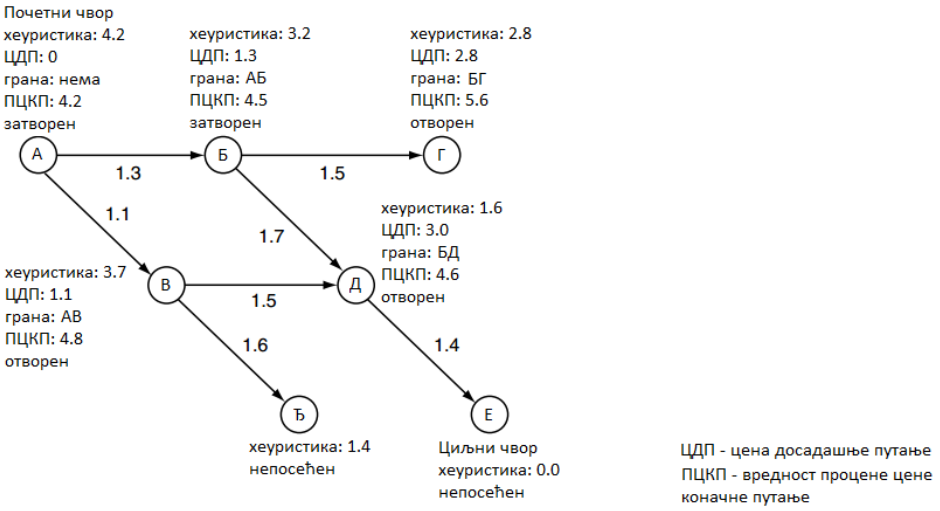
Током итерације, алгоритам A* разматра сваку грану ка даљим чворовима од тренутног чвора.

За сваку грану проналази се чвор који та грана повезује са тренутним чвором и смешта се тренутна збирна цена од почетног чвора (цена досадашње путање), као и грана од које се дошло до тог чвора, као и у случају Дијкстриног алгоритма.

Осим наведених података у случају алгоритма A* чува се још један додатан податак: вредност процене укупне цене коначне путање, уколико би она од почетног чвора ишла преко датог чвора до циљног чвора. Вредност процене укупне цене коначне путање можемо рачунати као збир цене досадашње путање и геометријске удаљености чвора од циљног чвора (која представља хеуристику у овом случају). Ова вредност процене рачуна се у одвојеном делу кода који није део алгоритма A*.

Процењене вредности удаљености чворова од циљног чвора називамо вредност хеуристике чвора. Вредност хеуристике чвора не може бити негативна (пошто су већ све цене у графу представљене ненегативним вредностима, нема смисла имати негативне вредности процена). Процена вредности хеуристке чворова од главног је значаја приликом имплементације алгоритма A*.

На слици 4.7 приказане су израчунате вредности хеуристике чворова као и цена досадашње путање и вредност процене укупне цене коначне путање.



Слика 4.7: A* са вредностима процене укупне цене коначне путање

Отворена и затворена листа чворова

Као и код Дијкстриног алгоритма, алгоритам A* има отворену и затворену листу чворова. И код алгоритма A* се у отвореној листи чворова чувају сви чворови за које алгоритам зна, али их још није обрадио кроз неку од досадашњих итерација. У затвореној листи чворова се чувају сви чворови које је алгоритам до датог тренутка обрадио у некој од итерација.

Чвор се убацује у листу отворених чворова у тренутку када је пронађен на крају неке од грана која до њега води од тренутног чвора. Чвор се пребацује у листу затворених чворова у тренутку када је обрађен у својој итерацији као тренутни чвор.

Код алгоритма A*, за разлику од Дијкстриног алгоритма се као наредни тренутни чвор у свакој од итерација уместо чвора са најнижом ценом досадашње путање одабира чвор са најнижом процењеном укупном ценом коначне путање. Често је то и различит чвор од оног са најнижом ценом досадашње путање.

Овакав начин одабира наредног чвора за разматрање даје предност чворовима који имају највише шансе да се нађу у коначној најкраћој путњи. Уколико чвор има ниску вредност укупне цене коначне путање, онда мора имати и релативно ниску вредност цене досадашње путање и релативно ниску вредност процењене удаљености до циљног чвора. Ако су процене тачне, чворови који су ближе циљном чвору ће се разматрати прво, чиме се значајно умањује простор претраге (постиже се нижа вредност испуне).

Израчунавање цене досадашње путање отворених и затворенних чворова

Током итерације могуће је наићи на чвор из отворене или затворене листе чворова као и код Дијкстриног алгоритма, за кога је потребно рекалкулисати податке.

Цена досадашње путање се израчунава на већ познати начин, а уколико је нова вредност нижа од постојеће вредности за дати чвор, потребно је ажурирати вредност на нову. Ажурирање података врши се само за цену досадашње путање (као једину поуздану вредност, пошто не садржи никакву процену), не и за укупну цену коначне путање.

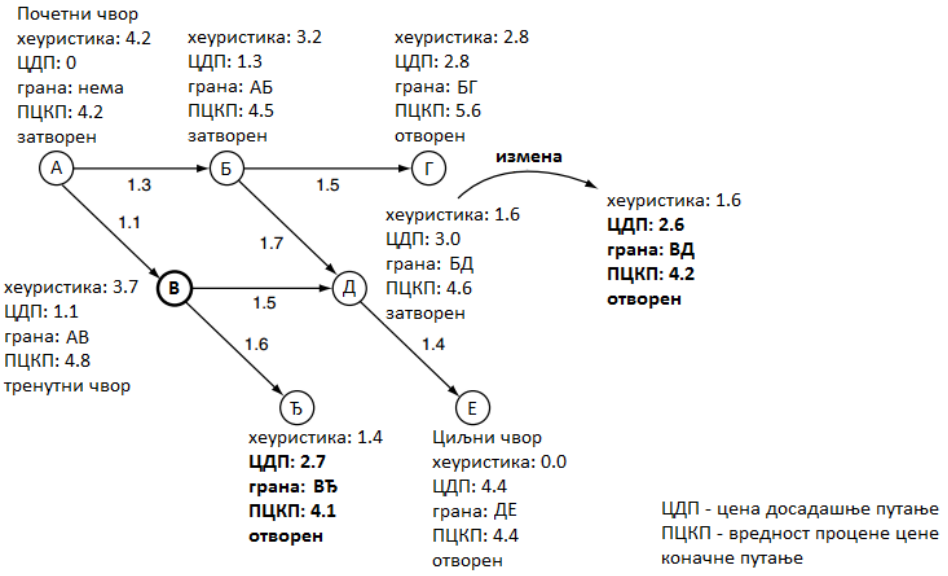
Код алгоритма A*, за разлику од Дијкстриног алгоритма, могуће је током даљег извршавања алгоритма пронаћи краће путање до чворова који се већ налазе у затвореној листи. Оваква ситуација настаје у случају уколико је ранијом оптимистичном проценом одабран чвор за који се сматрало да је најбољи одабир за дату итерацију иако он то заправо није био. Уколико је такав чвор смештен у затворену листу то значи да су све његове гране размотрене. То значи да постоји шанса да чворови на крајевима тих грана имају цену

досадашње путање засноване на цени досадашње путање датог чвора. Није довољно ажурирати само вредости за дати чвор. Потребно је проверити и вредности цене досадашње путање чворова који су гранама директно повезани са датим чвором и ажурирати их уколико је то потребно.

Ажурирање није потребно извршити ако се поново наиђе на чвор из отворене листе током извршавања алгоритма, зато што код таквих чворова нису обрађене све гране.

Постоји једноставан начин на који се алгоритам може принудити да изврши рекалкулисање и пропагирање нових вредности цене досадашње путање. То се постиже уклањањем датог чвора из затворене листе и његовим поновним смештањем у отворену листу.

На слици 4.8 приказан је случај када је потребно извршити рекалкулацију вредности цене досадашње путање одређеног чвора.



Слика 4.8: Ажурирање затвореног чвора

Нова путања до чвора Д, преко чвора В, је бржа, па је према томе ажуриран податак за чвор Д. Чвор Д је потребно убацити у листу отворених чворова. У наредној итерацији, пошто је чвор Д поново у отвореној листи, поново ће бити посећен и чвор Е где ће и за њега бити ажурирани подаци. То значи да ће чворови који ће бити поново посећени бити уклоњени из затворене листе чворова и поново убачени у отворену листу чворова.

Завршетак алгоритма

У многим имплементацијама, прекидање извршавања алгоритма A* врши се када циљни чвор има најнижу вредност процењене укупне вредности коначне путање.

Могуће је да поновно прерачунавање доведе до добијања још боље коначне путања, зато није добро прекинути извршавање алгоритма A* у том тренутку. Не може се гарантовати да је пронађена најкраћа путања само зато што циљни чвор има најнижу вредност у затвореној листи чворова. То значи да прекид извршавања алгоритма у моменту када циљни чвор има најнижу вредност у затвореној листи не гарантује да је пронађена најкраћа путања од почетног до циљног чвора.

Решење за овај случај јесте да се алгоритам A* пусти нешто дуже како би било гарантовано проналажење оптималног решења. То се може постићи тако што се захтева прекид извршавања алгоритма A* само у случају да чвор у отвореној листи са најнижом ценом досадашње путање има вишу вредност досадашње путање од цене путање која је пронађена за циљни чвор. Само у том случају може се гарантовати да ниједна накнадно пронађена путања неће формирати краћу путању до циљног чвора.

Имплементације алгоритма A* се у потпуности ослањају на чињеницу да је теоретски могуће добити не оптимално решење, што је могуће контролисати хеуристиком.

Зависно од одабира функције хеуристике, може се гарантовати оптимално решење или се намерно може дозволити добијање не оптималног решења како би се убрзало извршавање алгоритма. Уколико је процена дужине пута увек мања или једнака од стварне, онда алгоритам A* обавезно проналази најкраћи пут.

Дозвољавањем добијања не оптималног решења не постиже се велика уштеда у перформансама, али сматра се да се ипак свака ситница рачуна када је у питању побољшање перформанси игре.

Реконструкција најкраће путање

Реконструкција најкраће путање алгортима A* се врши на потпуно исти начин као и код Дијкстриног алгоритма. Почевши од циљног чвора у листу се купе

гране којим се дошло до циља корак по корак уназад. На крају се листа обрне како би се добила коначна најкраћа путања (листа грана које воде од почетног до циљног чвора).

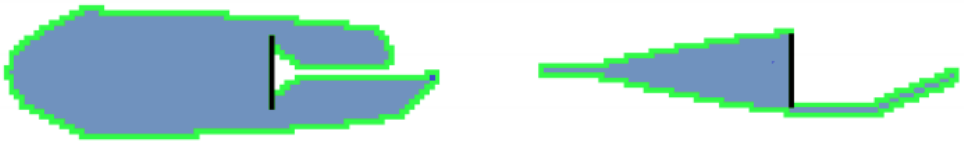
Функција хеуристике

Да би се обезбедила оптималност алгоритма A*, потребно је одабрати функцију хеуристике која не прецењује дужину путање од било ког чвора до циљног чвора у току претраге.

Коришћењем функције хеуристике која прецењује дужину путање до циљног чвора, алгоритам A* ће тежити да претражује чворове који су ближи циљном чвору. Добија се директнија претрага која ће брже ићи ка циљном чвору, али ће се тако претражити и мањи број чворова посебно оних који окружују почетни чвор што може лоше утицати на оприналност добијеног решења. Директнија претрага значи и нижу цену процесирања и мању искоришћеност меморије за проналажење најкраће путање.

Једина лоша страна коришћења функције која прецењује дужину путање до циљног чвора јесте што се губи оптималност алгоритма A*. Иако ће решење бити пронађено брже не гарантује се да ће оно бити и оптимално.

На слици 4.9 се јасно види разлика у испуњености простора претраге између функције хеуристике која подцењује и функције хеуристике која прецењује дужину путање до циљног чвора за исти проблем претраге.



Слика 4.9: Простор претраге коришћењем функције хеуристике која подцењује (лево) и функције хеуристике која прецењује (десно) дужину путање до циљног чвора

Варијанте алгоритма A*

Постоји велики број различитих варијанти алгоритма A*.

Свака од варијанти алгоритма A* има неку своју специфичности која побољшава основни алгоритам A* на одређени начин. Одабир боље хеуристике

и различите могућности оптимизација могу довести до додатног побољшања рада алгоритма A* када је у питању брзина израчунавања најкраће путање и број претражених чворова навигационог графа.

Неки од постојећих варијанти алгоритма A* су:

- ИДА* - A* алгоритам са итеративним продубљивањем (енг. Iterative Deepening A*).
- СМА* - поједностављење алгоритма МА* са ограниченим коришћењем меморије (енг. Simplified Memory-Bounded A*).
- Претрага ресицама - побољшање ИДА* алгоритма (енг. Fringe Search).
- ЛПА* - континуални алгоритам претраге, унапређење алгоритма A* (енг. Lifelong Planning A*).
- алгоритам Д* - динамички алгоритам A*.
- побољшани алгоритам Д* - (енг. D* Lite).

4.1.2.3 Код алгоритма A*

Као и код Дијкстриног алгоритма као аргументе алгоритам A* узима навигациони граф, почетни и циљни чвор.

Као четврти аргумент у овом случају узима се и објекат који може да изгенерише процену цене удаљености чворова од циљног чвора. Овај објекат представља коришћену хеуристику. Вредност коју алгоритам A* враћа након извршења представља листу грана које сачињавају најкраћу путању од почетног до циљног чвора.

```
1  def pathfindAStar(graph, start, end, heuristic):
2
3  # Struktura koja se koristi radi cuvanja
4  # porataka u o svakom od cvorova
5  struct NodeRecord:
6  node
7  connection
8  costSoFar
9  estimatedTotalCost
10
```

```
11 # Inicijalizacija podataka za pocetni cvor
12 startRecord = new NodeRecord()
13 startRecord.node = start
14 startRecord.connection = None
15 startRecord.costSoFar = 0
16 startRecord.estimatedTotalCost =
17 heuristic.estimate(start)
18
19 # Inicijalizacija otvorene i zatvorene liste cvorova
20 open = PathfindingList()
21 open += startRecord
22 closed = PathfindingList()
23
24 # Obrada cvorova kroz iteracije
25 while length(open) > 0:
26
27 # Pronaci element sa najnizom vrednoscu iz otvorene liste
28 # (posmatrajuci vrednost procene ukupne cene konacne putanje)
29 current = open.smallestElement()
30
31 # Ukoliko je trenutni cvor jednak ciljnom cvoru
32 # prekinuti izvr avanje algortima.
33 if current.node == goal: break
34
35 # U suprotnom pronaci grane ka drugim cvorovima od trenutnog cvora.
36 connections = graph.getConnections(current)
37
38 # Za svaku od datih grana trenutnog cvora...
39 for connection in connections:
40
41 # ... pokupiti vrednost procene ukupne cene
42 # cvorova povezanih tim granama sa trenutnim cvorom
43 endNode = connection.getToNode()
44 endNodeCost = current.costSoFar +
45 connection.getCost()
46
47 # Ukoliko je cvor zatvoren, potrebno ga je
48 # preskociti ili ukloniti iz zatvorene liste.
49 if closed.contains(endNode):
50
51 # Pronalaze se podaci iz zatvorene liste
52 # koji se odnose na ciljni  vor
53 endNodeRecord = closed.find(endNode)
54
```

```
55 # Ukoliko pronadjena putanja nije kraca, nastaviti dalje
56 if endNodeRecord.costSoFar <= endNodeCost:
57 continue;
58
59 # U suprotnom ukloniti ciljni cvor iz zatvorene liste
60 closed -= endNodeRecord
61
62 # Stare vrednosti cvora mogu se iskoristiti
63 # radi proracuna heuristike bez pozivanja
64 # potencijalno skupe funkcije heuristike
65 endNodeHeuristic = endNodeRecord.cost -
66 endNodeRecord.costSoFar
67
68 # Preskoci ako je cvor otvoren i
69 # nije pronadjena bolja putanja
70 else if open.contains(endNode):
71
72 # Pronalazenje podataka u otvorenoj listi
73 # koji odgovaraju ciljnom cvoru
74 endNodeRecord = open.find(endNode)
75
76 # Ukoliko putanja nije bolja, preskoci
77 if endNodeRecord.costSoFar <= endNodeCost:
78 continue;
79
80 # Stare vrednosti cvora mogu se iskoristiti
81 # radi proracuna heuristike bez pozivanja
82 # potencijalno skupe funkcije heuristike
83 endNodeHeuristic = endNodeRecord.cost -
84 endNodeRecord.costSoFar
85
86 # U suprotnom imamo neposecen cvor
87 # potrebno je popuniti podatke za njega
88 else:
89 endNodeRecord = new NodeRecord()
90 endNodeRecord.node = endNode
91
92 # Potrebno je izracunati vrednost heuristike
93 # koristeci funkciju, posto ne postoje stari
94 # podatci o cvoru koji bi mogli biti iskorisceni
95 endNodeHeuristic = heuristic.estimate(endNode)
96
97 # Ukoliko je potrebno azurirati podatke o cvoru
98 # Azurirati cenu, vrednost procene i granu
```

```
99  endNodeRecord.cost = endNodeCost
100 endNodeRecord.connection = connection
101 endNodeRecord.estimatedTotalCost =
102 endNodeCost + endNodeHeuristic
103
104 # I dodati cvor sa podacima u otvorenu listu
105 if not open.contains(endNode):
106 open += endNodeRecord
107
108 # Zavrsen prolazak kroz grane trenutnog cvora
109 # Dodati cvor u zatvorenu listu
110 # i ukloniti ga iz otvorene liste
111 open -= current
112 closed += current
113
114 # Ukoliko trenutni cvor nije ciljni cvor
115 # ili kada nema vise cvorova za pretragu
116 if current.node != goal:
117
118 # Ponestalo je cvorova, a resenje nije pronadjeno
119 return None
120
121 else:
122
123 # Napraviti listu u koju je
124 # potrebno smestiti konacnu putanju
125 path = []
126
127 # Vracajuci se unazad formirati konacnu putanju,
128 # sakupljajuci grane u listu
129 while current.node != start:
130 path += current.connection
131 current = current.connection.getFromNode()
132
133 # Obrnuti listu kako bi sadzala grane
134 # konacne putanje sortirane od pocetnog
135 # ka ciljnom cvoru.
136 # Vratiti listu kao rezultat.
137 return reverse(path)
```

Глава 5

Примери и тестови

За развој примера и тестова коришћен је Јунити 5.6.0 (енг. Unity) као развојно окружење. Јунити је вишеплатформско развојно окружење. Користи се за развој видео игара за веб, десктоп платформе, конзоле и мобилне уређаје. Развијен је 2005. године од раније коришћеног покретача игре (енг. game engine) OS X-а у вишеплатформски покретач игре.

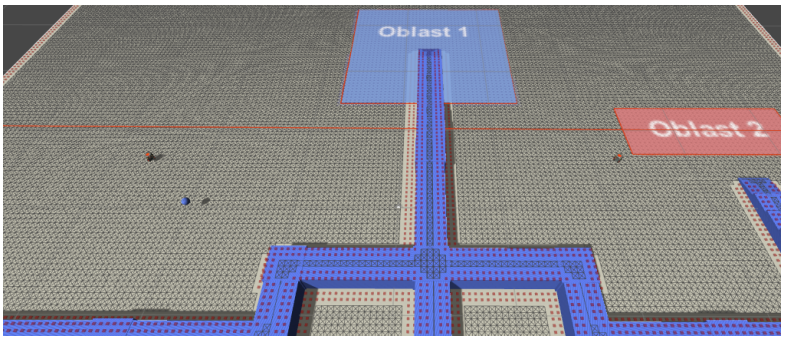
Од могућих навигационих графова за коришћење у тестовима одабрани су навигациони графови засновани на мрежи, зато што се они користе код већине савремених видео игара.

У тачки 5.1 приказани су реализовани начини прадстављања мапа. Два реализована алгоритма (Дијкстрин алгоритам и алгоритам A*) упоређена су у обради статичке сцене у тачки 5.2. Експеримент са проналажењем најкраћег пута на мапи са динамичким променама приказан је у тачки 5.3.

5.1 Примери начина представљања мапа

Направљено је неколико примера начина представљања мапа:

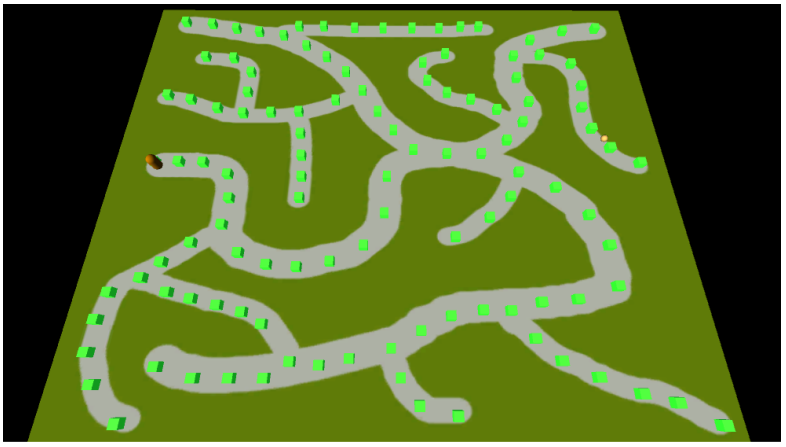
- навигационим графом заснованом на мрежи.
- навигационим графом заснованим на маркерима.
- навигационим графом заснованим на мрежама конвексних полигона.



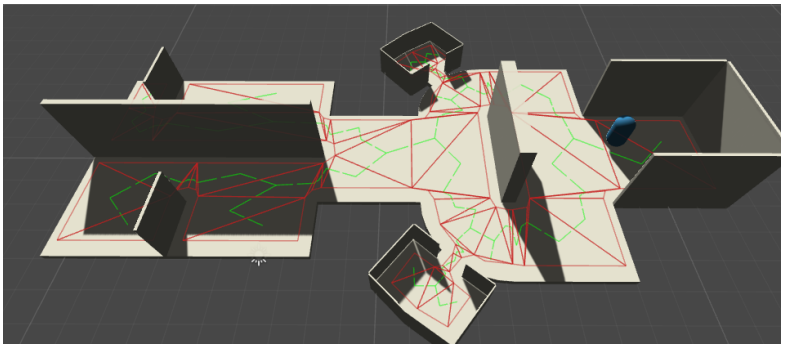
Слика 5.1: Мапа са навигационим графом заснованом на мрежи.

Додатно на сцени са навигационим графом заснованом на мрежи направљен је и пример пролазне и непролазне области, као и пример додељивања негативних бодова приликом преласка робота преко неке од области.

На овај начин се неким објектима може забранити или омогућити уз одређене пенале (успорење) прелазак преко воде и слично.



Слика 5.2: Мапа са навигационим графом заснованом на маркерима.



Слика 5.3: Мапа са навигационим графом заснованим на мрежама конвексних полигона.

Примери су урађени тако да се за претрагу може користити Дијкстрин алгоритам или алгоритам A*.

5.2 Поређење Дијкстриног алгоритма и алгоритма A*

Како би се видела разлика у брзини рада и броју претражених чворова између Дијкстриног алгоритма и алгоритма A* направљен је пример који јасно указује на разлике између ова два алгоритма.

Очекивано је да ће резултујућа путања бити једнаке дужине коришћењем било којег од наведена два алгоритма, али и да ће алгоритам A* претражити далеко мање чворова навигационог графа и утрошити мање времена за добијање резултујуће путање.

Дијкстрин алгоритам сигурно проналази најкраћу путању, а такође под одређеним условима и алгоритам A* гарантовано проналази најкраћу путању. Може се десити да алгоритам A* не пронађе најкраћу путању у случају да је одабрана лоша хеуристика.

5.2.1 Опис теста

Тест је урађен коришћењем три сцене. Основна разлика између сцена јесте у густини објеката на мапи. Свака од сцена садржи три нивоа који се разликују по димензијама мапе (по броју чворова у навигационом графу). Димензије коришћених мапа су 200×200, 300×300 и 500×500 чворова навигационог графа заснованог на мрежи. Омогућено је кретање у осам праваца (сваки од чворова повезан је са осам суседа осим крајњих чворова и чворова око објеката на мапи).

Циљ је показати разлику у простору претраге и потребном времену за проналажење најкраће путање између Дијкстриног алгоритма и алгоритма A*, као и како густина објеката на мапи и величина мапе утичу на различите параметре у претрази.

Генерисање објеката на мапи врши се насумичним одабиром између осам различитих објеката који се постављају на унапред дефинисаним местима на мапи. Места генерисања су унапред дефинисана како не би дошло до проблема са преклапањем изгенерисаних објеката на мапи. Од постављене густине објеката на мапи зависи колико ће и на којим местима бити изгенерисано објеката. Са повећањем густине мапе увећава се и број места генерисања објеката на локацијама које ће утицати на кретање робота. Дијкстрин алгоритам и алгоритам A* су при истој величини мапе и при истој густини објеката на мапи покретани при једнако изгенерисаној поставци објеката како би се добили упоредиви резултати и једнака резултујућа путања.

Параметри у претрази који се узимају у обзир су број претражених чворова навигационог графа, дужина добијене путање и време потребно за израчунавање добијене путање. Покренуто је девет тестова са раличитим поставкама на мапи (различита димензија мапе и густина објеката на мапи) коришћењем Дијкстриног алгоритма и исти број тестова са датим поставкама коришћењем алгоритма A*.

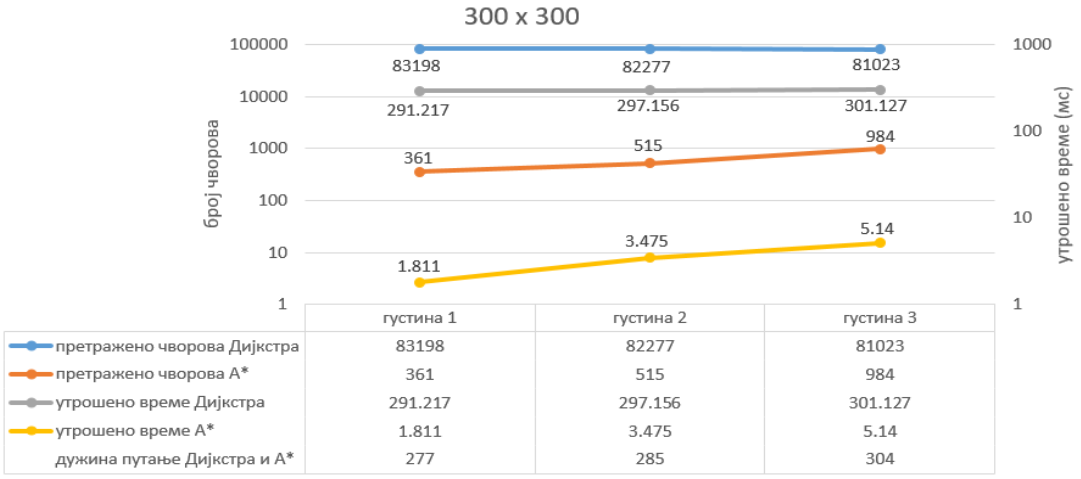
5.2.2 Резултати теста

Густина 1 у резултатима подразумева да је коришћено девет објеката, густина 2 тринаест објеката, а густина 3 петнаест објеката.

Добијени су следећи резултати за различите димензије мапе:

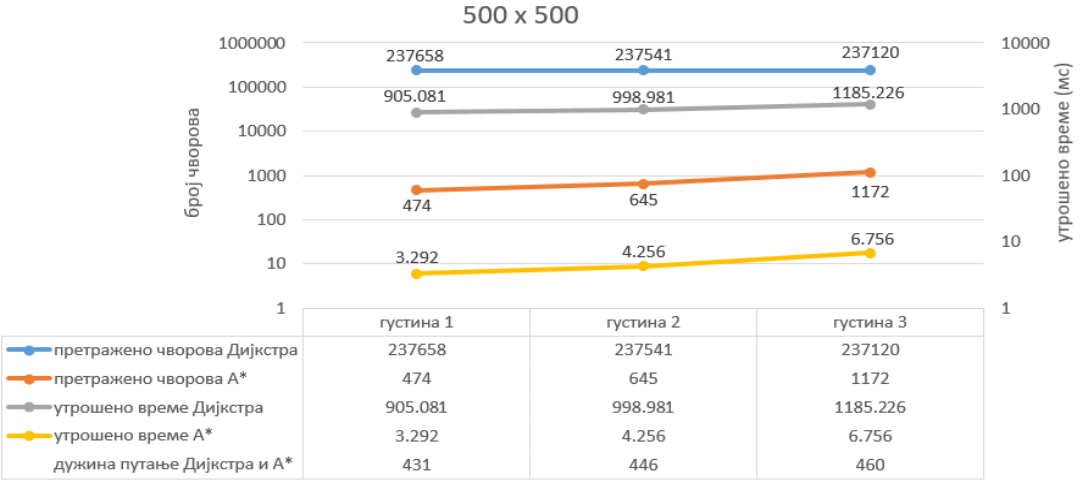
	густина 1	густина 2	густина 3
претражено чворова Дијкстра	34078	32785	31715
претражено чворова A*	297	311	409
утрошено време Дијкстра	121.128	123.775	128.126
утрошено време A*	1.489	2.459	2.691
дужина путање Дијкстра и A*	185	190	201

Слика 5.4: Добијени резултати за димензију мапе 200*200.



	густина 1	густина 2	густина 3
претражено чворова Дијкстра	83198	82277	81023
претражено чворова A*	361	515	984
утрошено време Дијкстра	291.217	297.156	301.127
утрошено време A*	1.811	3.475	5.14
дужина путање Дијкстра и A*	277	285	304

Слика 5.5: Добијени резултати за димензију мапе 300*300.



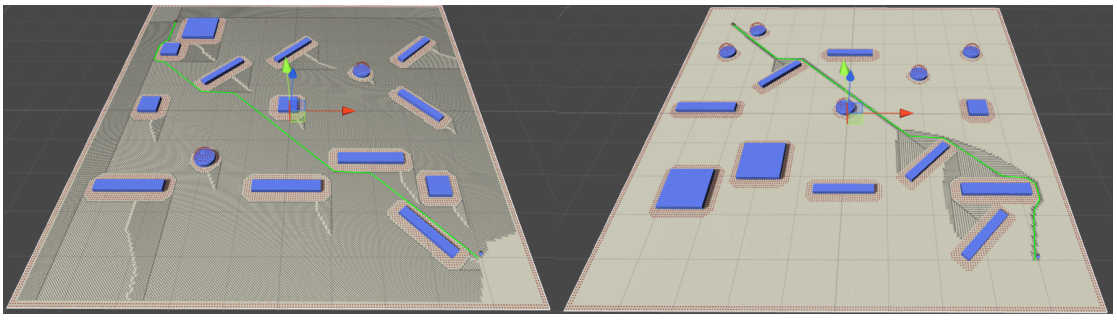
	густина 1	густина 2	густина 3
претражено чворова Дијкстра	237658	237541	237120
претражено чворова A*	474	645	1172
утрошено време Дијкстра	905.081	998.981	1185.226
утрошено време A*	3.292	4.256	6.756
дужина путање Дијкстра и A*	431	446	460

Слика 5.6: Добијени резултати за димензију мапе 500*500.

5.2.3 Анализа резултата теста

Јасно се уочава разлика када је у питању број претражених чворова и потребно време за израчунавање путање између Дијкстриног алгоритма и алгоритма A*. Када је у питању дужина путање она је за исту величину мапе и исту густину објеката на мапи једнака било да је коришћен Дијкстрин алгоритам или алгоритам A*, што је постигнуто начином на који су изгенерисани објекти на мапи и добрим одабиром хеуристике алгортима A*.

Са увећањем мапе увећавају се и количина утрошеног времена и број претражених чворова, а приметно је и увећање количине утрошеног времена и броја претражених чворова у тесту и са порастом густине објеката на мапи. Изузетак постоји код Дијкстриног алгоритма где се за било коју димензију



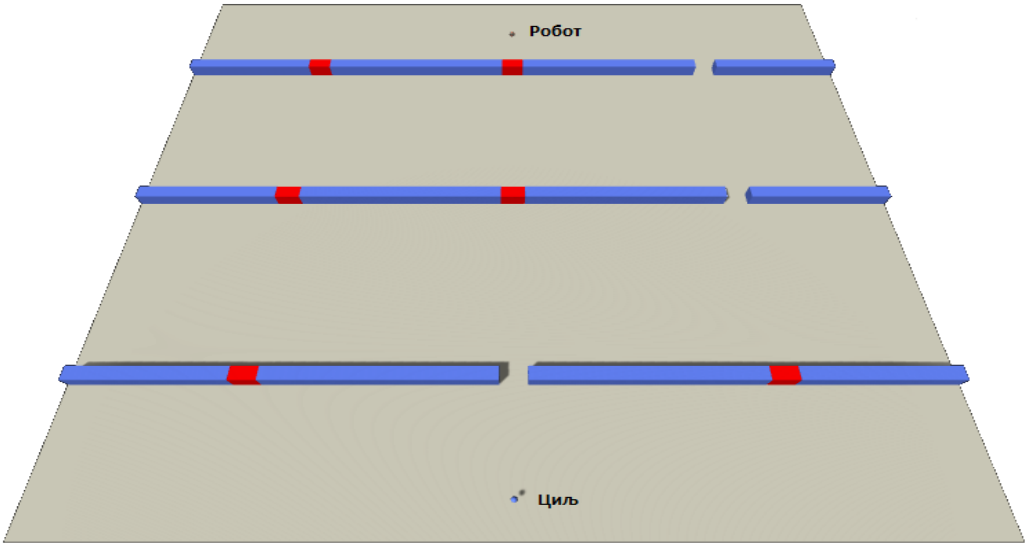
Слика 5.7: Поређење испуњености простора претраге коришћењем Дијкстриног алгоритма (лево) и коришћењем алгоритма A* (десно).

мапе број претражених чворова за нијансу умањује, зато што се повећањем густине објеката на мапи повећава и број непрелазних чворова.

5.3 Пример динамичке промене мапе

5.3.1 Опис примера

Примером динамичке промене мапе показано је да и поред динамичких промена на мапи робот проналази пут до свог циља. Пример је урађен тако да се за претрагу може користити Дијкстрин алгоритам или алгоритам A*.



Слика 5.8: Пример динамичке промене мапе са зидовима и вратима.

На сцени се налазе три зида, а на сваком од зидова налазе се троја врата. Омогућено је отварање по једних врата на сваком од зидова. Насумичном

изменом распореда отворених врата робот проналази нову најкраћу путању до циља на мапи.

Постоји могућност постављања и уклањања препрека на мапи са аутоматским ажурирањем прелазних и непрелазних чворова у графу приликом постављања објекта.

Глава 6

Закључак

Основна идеја овог рада је представљање алгоритама који се користе за проналажење најкраће путање у видео играма, као и њихово упоређивање.

Разматрајући различите врсте игара које данас постоје уочено је да реализација већине данашњих игара зависи од успешности кретања карактера кроз окружење игре, што значи да је систем за проналажење најкраћег пута један од основних система вештачке интелигенције који би морала имати свака данашња видео игра.

Услед високог нивоа сложености окружења модерних видео игара, како би проналажење најкраће путање коришћењем неког од алгоритама било могуће, неопходно је представљање мапа на најједноставнији могући начин користећи навигационе графове. Тиме се постиже уштеда процесорског времена и меморијског простора.

Уобичајена је примена три погодне апстракције комплексних тродимензионалних окружења модерних игара: навигациони графови засновани на мрежи, на маркерима и на мрежама конвексних полигона. Одабир неке од ових апстракција зависи од нивоа сложености окружења видео игре.

Показана је велика разлика између алгоритма A* и Дијкстриног алгоритма. Алгоритам A* показао се доста бољим алгоритмом за коришћење у видео играма које се данас користе, због брзине израчунавања крајње путање и малог броја претражених чворова навигационог графа.

У даљем раду могуће је детаљније обрадити различите варијанте алгоритма A* као и детаљније обрадити различите хеуристике и могућности оптимизација које могу довести до додатног побољшања рада алгоритма A* када је у питању брзина израчунавања најкраће путање и број претражених чворова навигационог графа.

Прилог: речник термина

Кретање у јату (енг. flocking)

Основни модели кретања у јату изводе се коришћењем три једноставна правила:

- Раздвајање - избегавање груписања велике количине суседних јединки.
- Поравнање - кретање ка просечном правцу суседа.
- Кохезија - кретање ка просечној позицији суседа.

Граф

Граф је апстрактни математички објекат, а цртеж који се састоји од тачака и линија је само геометријска представа графа. Међутим, уобичајено је да се таква слика назива графом.

Полигон

Полигон је фигура у равни коју чини многоугаона линија и унутрашња област одређена том линијом. Други назив је многоугао.

Вештачка интелигенција (енг. artificial intelligence)

Вештачка интелигенција бави се, превасходно, проблемима у којима се јавља комбинаторна експлозија, проблемима чије решавање захтева разматрање огромног броја могућности. Решавање таквих проблема обично се своди на неку врсту претраге, систематичног поступка обраде великог броја могућности.

Механика извођења игре (енг. gameplay)

Односи се на одређени начин интеракције између играча и игре, а посебно када је у питању видео игра. Механика извођења игре представља шему дефинисану правилима игре, начинима интеракције између играча и игре, изазовима и превазилажењем истих, заплетом игре и повезаношћу играча са њим.

Атомична акција

У програмирању атомична акција је она која се ефективно извршава целокупна одједном. Атомична акција се не може зауставити усред извршавања или се целокупна извршава истовремено или се уопште и не изврши. На пример, наручивање авионске карте преко интернета захтева извршење две акције: плаћање и резервацију седишта. Потенцијални путник или резервише седиште и плаћа резервацију или нити резервише седиште и не плати резервацију.

Фрејм

Приликом приказивања снимака приказује се одређени број слика у секунди. Једна таква слика која се користи за приказ снимка назива се фрејм. Брзина смењивања слика или број слика по секунди (енг. FPS - frames per second) мера је фреквентности промене слика, односно колико се слика приказује у секунди. Термин се односи и на филмове, видео камере, рачунарску графику.

Време одзива (енг. response time)

Време одзива је време потребно да систем или функционална јединица одреагује на дати улаз.

Емпирија

Практично искуство; знање, засновано на научним чињеницама, искуство, проучавање путем посматрања чињеница, наука о искуству.

Итеративни алгоритам

Итеративни алгоритми су алгоритми који за разлику од рекурзивних алгоритама не позивају самог себе већсе ослањају на структуре попут петљи и додатне структуре података као што је низ или ред да би решили проблем.

Хеуристика

Хеуристика обухвата методе и технике решавања проблема, учења и откривања који су базирани на искуству. Хеуристички методи се користе да убрзају процес проналажења довољно доброг решења у ситуацијама када спровођење детаљног истраживања није практично. Примери тога обухватају коришћење разних уопштених правила, информисаног нагађања, интуиције и здравог разума.

Библиографија

[1] Ian Millington; John Funge. In *Artificial intelligence for games second edition*. Morgan Kaufmann, Burlington, MA, 2009.

[2] Mat Buckland. In *Programming Game AI by Example*. Wordware Publishing, Inc., Plano, TX, 2005.

[3] Steve Woodcock. Flocking: A simple technique for simulating group behavior. In Mark DeLoura, editor, *Best of Game Programming Gems.*, pages 297–310. Course Technology, Cengage Learning, Boston, MA, 2008.

[4] Steve Rabin. A* aesthetic optimizations. In Mark DeLoura, editor, *Game Programming Gems*, pages 264–271. Charles River Media, Boston, MA, 2008.

[5] Johnson Geraint. Smoothing a navigation mesh path. In Steve Rabin, editor, *AI Game Programming Wisdom 3*, pages 129–140. Charles River Media, Boston, MA, 2006.

[6] Pinter Marco. Realistic turning between waypoints. In Steve Rabin, editor, *AI Game Programming Wisdom*, pages 186–192. Charles River Media, Boston, MA, 2002.

[7] A. Botea; M. Muller; J. Schaeffer. Near-optimal hierarchical pathfinding. *Journal of Game Development*, 1, September 2006. URL https://webdocs.cs.ualberta.ca/~mmueller/ps/hpastar.pdf.

[8] Chakrabarti. Heuristic search in restricted memory. pages 197–221. Artificial Intelligence Journal (AIJ), Elsevier, 1989.

[9] Xiao Cui; Hao Shi. A*-based pathfinding in modern computer games. *IJCSNS International Journal of Computer Science and Network Security*, 11, January 2011. URL http://paper.ijcsns.org/07_book/201101/20110119.pdf.