Wydział Podstawowych Problemów Techniki
Politechnika Wrocławska

Przegląd i porównanie algorytmów wyszukiwania ścieżek w grach komputerowych

Jakub Kołakowski
Nr indeksu: 221457

Praca inżynierska napisana
pod kierunkiem
dr. inż. Marcina Zawady

Politechnika
Wrocławska

Wrocław 2017

Spis treści

Wstęp

Praca swoim zakresem obejmuje problem znajdowania najkrótszej ścieżki. Jest on widoczny w wielu dziedzinach takich jak robotyka, zarządzanie ruchem oraz gry komputerowe. Praca skupia się na ostatniej z powyższych dziedzin. W nowoczesnych grach strategicznych, na mapie może znajdować się wiele setek jednostek jednocześnie. Każda z nich może w dowolnym momencie zostać skierowana przez gracza w arbitralne położenie na mapie. W celu zapewnienia płynności rozgrywki oraz wysokiej jakości satysfakcji związanej z produktem, muszą zostać użyte zaawansowane algorytmy, pozwalające na wyznaczenie wysokiej jakości trasy na każde żądanie gracza. Dodatkowo narzucone są regorystyczne wymagania, jakimi są: określony czas, w którym musi zostać znaleziona ścieżka, ograniczona dostępność pamięci oraz odpowiednia jakość otrzymanego rozwiązania.

Celem pracy jest przeanalizowanie i porównanie wybranych algorytmów rozwiązujących problem znajdowania najkrótszej ścieżki, w kontekście gier komputerowych. Wybrane algorytmy zostaną zaimplementowane, a następnie poddane eksperymentom doświadczalnym na trzech, mocno zróżnicowanych zbiorach map. [15]

Praca składa się z czterech rozdziałów. W pierwszym przedstawiony został dokładniej problem znajdowania najkrótszej ścieżki. Opisano ograniczenia towarzyszące temu zagadnieniu. Przedstawione zostało kilka sposobów wewnętrznej reprezentacji świata rozgrywki oraz wytłumaczone pojęcia używane w dalszej części pracy.

Drugi rozdział zawiera dokładną analizę wybranych algorytmów. Są nimi A*, Hierarchical Pathfinding A* (HPA*), Jump Point Search(JPS) oraz Fringe Search(FS). Dla każdego z algorytmów podany został pseudokod wraz z opisem działania oraz wyjaśnione zostały ważne kwestie związane z tymi algorytmami. Wzięte pod uwagę zostały ograniczenia nałożone na algorytmy.

W trzecim rozdziale przedstawione zostały wyniki przeprowadzonych eksperymentów. Wyniki zostały omówione, a na ich podstawie porównane zaimplementowane algorytmy. Przedstawione zostały mocne i słabe strony każdego z algorytmów na każdym z użytych zbiorów map.

Rozdział czwarty opisuje technologie użyte podczas implementacji projektu. Przedstawiony jest szczegółowy diagram klas. Rozdział zawiera dokumentację techniczną części kodów źródłowych poszczególnych części systemu.

Końcowy rozdział podsumowuje wyniki przeprowadzonych eksperymentów oraz przedstawia możliwe ścieżki dalszego rozszerzania projektu.

Analiza problemu

2.1 Ograniczenia

Problem znalezienia ścieżki w grach komputerowych sprowadza się do znalezienia możliwej do pokonania i najlepiej jak najkrótszej trasy między dwoma lub większą ilością miejsc na mapie. W strategiach czasu rzeczywistego (RTS), każda jednostka w grze, może zostać pokierowana przez gracza z jednego punktu na mapie do dowolnego innego. W międzyczasie sztuczna inteligencja może rozkazać innym jednostkom patrolowanie pewnego obszaru. Skala problemu rośnie po narzuceniu dodatkowych ograniczeń, wśród których można wyróżnić trzy najbardziej znaczące: ograniczona ilość czasu na znalezienie ścieżki, ilość dostępnej pamięci oraz oczekiwana optymalność długości ścieżki. Należy również wziąć pod uwagę, że wyszukiwanie może odbywać się przez wiele jednostek jednocześnie oraz w zmiennym środowisku, co może powodować, że ścieżka przestaje być możliwa do przejścia i konieczne jest wyznaczenie nowej. Pokrótce przybliżone zostanie każde z tych ograniczeń.

Aby gra wyglądała płynnie dla użytkownika powinna działać przynajmniej w 30 klatkach na sekundę (FPS). Wiele gier, aby jeszcze bardziej zwiększyć realizm rozgrywki działa w 60 FPS. Daje to kolejno około 33ms i 16ms czasu między kolejnymi iteracjami pętli gry. Czas ten dzielony jest na kilka elementów, jak wyrenderowanie aspektów wizualnych, wykrycie kolizji czy aktualizacja logicznej części środowiska gry. Mimo przeniesienia większości nakładu pracy związanej ze sferą wizualną na karty graficzne (GPU), sztucznej inteligencji zostaje maksymalnie 20% czasu z każdej iteracji, daje to kolejno około 6,6ms oraz 3,2ms dostępnego czasu procesora. [10] Niespełnienie tego regorystycznego warunku powoduje widoczne opóźnienia w reakcji postaci na żądania gracza.

Kolejnym ograniczeniem jest ilość maksymalnej dostępnej pamięci przydzielonej dla problemu wyszukiwania ścieżki. Na dzień dzisiejszy problem ma mniejsza wagę z powodu dostępności dużej ilości pamięci zarówno na platformach PC, jak i konsolach. Problem był bardziej dostrzegalny kiedyś. Dla przykładu gra Brutal Legend skierowana na konsole podczas wyszukiwania miała dostęp maksymalnie do 6MB pamięci. [10]

Ostatnim z powyżej wymienionych ograniczeń jest optymalność znalezionej ścieżki. W grach komputerowych ma ona niższy priorytet. Ważne jest, aby ścieżka wyglądała zadowalająco dla gracza, możliwe są więc kilkuprocentowe odchyły od optimum. Z tego powodu często używane są algorytmy zwracające jedynie przybliżone rozwiązanie. [8]

2.2 Środowisko

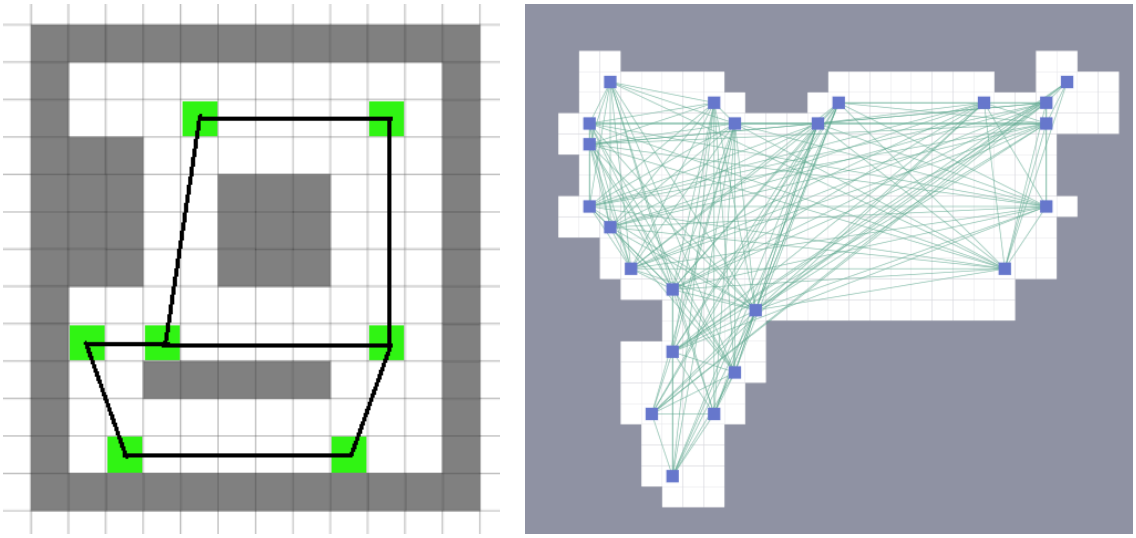
Znajdowanie ścieżki zwykle definiowane jest w środowisku 2D. Nawet dla gier 3D, prawie zawsze możliwa jest redukcja do problemu 2D, z uwagi na to, że ruch odbywa się po ziemi. Z tego powodu wiele klasycznych algorytmów definiowanych jest tylko dla przypadków dwuwymiarowych. [12] Jednostki zwykle zajmują określoną ilość przestrzeni na mapie. Może to zostać uwzględnione już na etapie wyznaczania ścieżki, bądź później przy pomocy detektorów kolizji. Wyznaczanie ścieżki nie odbywa się bezpośrednio na geometrycznej reprezentacji mapy. Ukształtowanie terenu zostaje przybliżone, poprzez sprowadzanie wewnętrznego sposobu reprezentacji mapy gry do grafu. Graf jest matematyczną strukturą reprezentowaną przez zbiór wierzchołków i łączących je krawędzi. Każdej z krawędzi może zostać przypisana dodatkowa informacja określana mianem wagi, która w kontekście gier określa czas, odległość, bądź też koszt podróży między konkretnymi dwoma wierzchołkami. Znalezienie ścieżki sprowadza się do znalezienia kolejnych, sąsiednich wierzchołków, których pokonanie umożliwi podróż z punktu startowego do celu. Wierzchołki uznawane są za sąsiednie,

jeśli łączy je krawędź. Najkrótszą ścieżką jest lista wierzchołków, których sumaryczna waga krawędzi jest zminimalizowana, czyli nie istnieje inna ścieżka, którą można dotrzeć szybciej do celu.

Istnieje kilka popularnych wyborów reprezentacji mapy. Są nimi punkty nawigacyjne, siatki nawigacyjne, mapy wielokątów, reprezentacje hierarchiczne oraz krata. Każda ze struktur posiada zarówno korzyści, jak i słabe strony. W szczególności krata ośmiokątna cechuje się prostotą w implementacji, ale jednocześnie narzuca spory czynnik rozgałęzienia, który dla każdego wierzchołka jest mniejszy, bądź równy 8. Oznacza to, że podczas rozpatrywania danego wierzchołka, algorytm musi wziąć pod uwagę maksymalnie 8 jego sąsiadów.

Kilka z powyższych struktur zostanie pokrótce omówiona.

2.2.1 Punkty nawigacyjne



(a) Punkty nawigacyjne (b) Automatyczna generacja punktów widoczności [9]

Rysunek 2.1

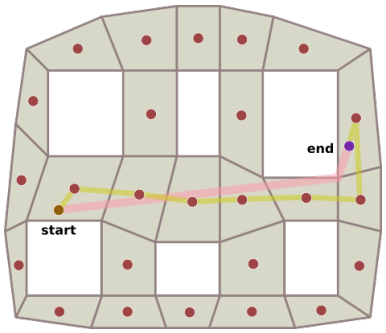
Reprezentacja mapy poprzez punkty nawigacyjne sprowadza się do wyboru charakterystycznych punktów, w których zachodzi duże prawdopodobieństwo zmiany kierunku podróży. Punkty są tak dobierane, aby każdy wierzchołek był połączony z przynajmniej jednym, innym wierzchołkiem znajdującym się na linii widoczności (rys 2.1.a). Mogą one zostać wstawione ręcznie przez osobę projektującą dany poziom rozgrywki, jednak proces ten może być czasochłonny. Inną możliwością jest automatyczne wygenerowanie punktów nawigacyjnych i łączących je krawędzi. Problemem wtedy staje się zoptymalizowanie ilości wygenerowanych krawędzi. Sytuacja przedstawiona jest na rysunku 2.1.b. Poprawnie stworzony graf pozwala na zmniejszenie rozpatrywanej przestrzeni stanów podczas wyszukiwania.

2.2.2 Siatka nawigacyjna

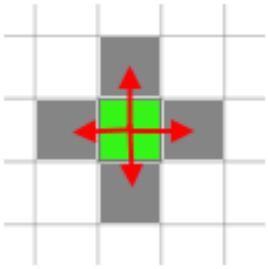
Siatki nawigacyjne pozwalają na reprezentowanie poszczególnych obszarów map przez wielokąty. Do każdego z obszarów może zostać przypisana dodatkowa informacja charakteryzująca go, na przykład zaklasyfikowanie terenu jako drogi, lasu, bądź rzeki. Każdy z wielokątów może być następnie traktowany jako pojedyncza płytka. Przykład siatki nawigacyjnej przedstawia rysunek 2.2.

2.2.3 Kraty

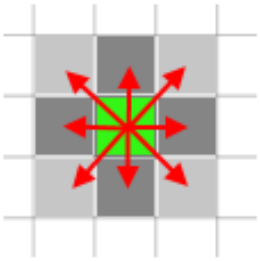
Kraty dzielą się na kilka rodzajów, w zależności od możliwych ruchów między sąsiednimi kratami. Są to kwadraty(rys 2.3), sześciokąty(rys 2.4) oraz ośmiokąty(rys 2.5). Każda krata reprezentowana jest jako wierzchołek, a połączenie jako krawędź między sąsiednimi wierzchołkami. W pracy wybrana została reprezentacja



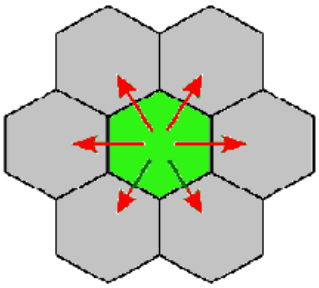
Rysunek 2.2: Siatka nawigacyjna [6]



Rysunek 2.3: Mapa kwadratowa



Rysunek 2.4: Mapa ośmiokątna



Rysunek 2.5: Mapa sześciokątna

w postaci kraty ośmiokątnej z uwagi na szerokie wykorzystanie, w szczególności w grach typu RTS, oraz na prostotę tejże struktury. Umożliwia ona poruszanie się łącznie w 8 stron, 4 po prostej i 4 po przekątnej.

2.3 Teoria

Omówione tutaj zostaną pojęcia, używane w dalszej części pracy.

Sąsiedzi wierzchołka - wierzchołki przylegające do aktualnie rozpatrywanego wierzchołka, do których można dostać się bezpośrednio, za pomocą jednego ruchu.

Odwiedzenie wierzchołka - podjęcie decyzji o wstawieniu wierzchołka do otwartej listy, w celu dalszego rozpatrzenia, bądź o tym czy dany wierzchołek ma zostać rozpatrzony w danej iteracji algorytmu.

Rozpatrzenie wierzchołka - przeanalizowanie wszystkich sąsiadów danego wierzchołka, aktualizacja najkrótszych ścieżek do nich, oraz uaktualnienie stanu otwartej i zamkniętej listy.

Otwarta lista - składa się z wierzchołków, które zostały odwiedzone, ale jeszcze nie rozpatrzone. Z nich wybierany jest kolejny wierzchołek do rozpatrzenia.

Zamknięta lista - trafiają do niej wierzchołki, które zostały odwiedzone oraz rozpatrzone.

Najkrótsza(optymalna) ścieżka - lista wierzchołków, których sumaryczna waga krawędzi jest zminimalizowana, czyli nie istnieje inna ścieżka, którą można dotrzeć szybciej do celu.

Funkcja heurystyczna h(n) - funkcja zwracająca oszacowanie odległości od wierzchołka n do celu.

Funkcja g(n) - funkcja określająca odległość wierzchołka n od punktu startowego

Funkcja ewaluacji f(n) - jej wzór to $f(n) = g(n) + h(n)$. Stanowi podstawę uporządkowania elementów w otwartej liście.

Wygładzanie ścieżki - Zależnie od kontekstu jest to albo próba zastąpienia nieoptymalnych fragmentów wyznaczonej ścieżki, krótszymi, albo zastąpienie nierealistycznie wyglądających fragmentów ścieżki, jak nagłe skręty pod kątem prostym, bardziej atrakcyjnymi wizualnie dla użytkownika.

Czynnik rozgałęzienia - w kontekście problemu wyszukiwania ścieżek jest to średnia ilość sąsiadów przypadająca na każdy wierzchołek

Łamanie remisów - metoda pozwalająca na rozróżnienie wierzchołków z tą samą f-wartością, ale różnymi g i h-wartościami.

Algorytmy i ich porównanie

3.1 Algorytm A*

Chcąc dostać się z jednego miejsca na mapie do drugiego istotne jest znalezienie jak najkrótszej ścieżki. Kolejnym wymogiem jest skrócenie etapu planowania podróży. Aby tego dokonać, algorytm w każdym kroku musi podejmować decyzję, który wierzchołek należy rozpatrzyć jako następny, aby zminimalizować oba te czynniki. Algorytm obrazuje pseudokod 3.1.

Pseudokod 3.1: Pseudokod dla algorytmu A*

```
1  Insert start to OPEN list
2  while OPEN not empty do
3      find node N with lowest f-value in OPEN list
4      pop N from OPEN list
5      if N == goal then
6          return found path
7      foreach S ∈ Successors(N) do
8          g_new(S) = g(N) + weight(S,N)
9          if S is in OPEN list and g(S) > g_new(S) then
10             g(S) = g_new(S)
11             parent(S) = N
12         else if S is in CLOSED list and g(S) > g_new(S) then
13             g(S) = g_new(S)
14             parent(S) = N
15             pop S from CLOSED list
16             add S to OPEN list
17         else if S is not expanded then
18             g(S) = g_new(S)
19             parent(S) = N
20             calculate h(S)
21             add S to OPEN list
22     add N to CLOSED list
23 return path not found
```

3.1.1 Opis algorytmu

Algorytm działa w sposób iteracyjny. Podczas każdej iteracji rozpatruje on kolejne wierzchołki grafu, które uzna za najbardziej prawdopodobne znalezienia się na najkrótszej ścieżce. Za rozpatrzenie wierzchołka uznaje się sprawdzenie połączeń do wszystkich sąsiednich wierzchołków, w celu aktualizacji informacji o ich położeniu. Dla każdego sąsiada sprawdzane jest czy należy on do otwartej, bądź zamkniętej listy. W otwartej liście znajdują się wierzchołki do rozpatrzenia w przyszłości, w zamkniętej te, które już zostały odwiedzone i uznane za rozpatrzone. Dla każdego wierzchołka przechowywana jest informacja o jego odległości od punktu startowego oraz informacja o rodzicu, z którego bezpośrednio można dostać się do aktualnie rozpatrywanego wierzchołka. Dodatkowo można przechowywać heurystyczne oszacowanie jego odległości od celu. Pozwala to na uniknięcie tych samych obliczeń podczas ponownego odwiedzania wierzchołka, jednak zwiększa zużycie

pamięciowe algorytmu. Dla każdego z sąsiadów, w razie odkrycia krótszej od dotychczas wyznaczonej ścieżki do danego sąsiada, nastepuje aktualizacja informacji o jego położeniu. Nowy koszt podróży można obliczyć wykorzystując znany już dotychczasowy koszt podróży do rodzica oraz koszt między rodzicem, a aktualnie rozpatrywanym wierzchołkiem.

Jeśli sąsiad jest już w otwartej liście, następuje jedynie aktualizacja informacji o nim, natomiast jeśli w zamkniętej, oznacza to, że znaleziona została inna, bardziej optymalna ścieżka, trzeba więc ponownie rozpatrzyć jego sąsiednie wierzchołki. Pojawia się problem wielokrotnego rozpatrywania tych samych wierzchołków. Spowodowany jest on kolejnością odwiedzania wierzchołków z otwartej listy. Problem nie występuje w algorytmie Dijkstry, gdzie w każdej iteracji z otwartej listy zdejmowany jest wierzchołek aktualnie najbliżej punktu startowego. W A* o tej kolejności decyduje dodatkowo oszacowanie jego odległości do celu. Ostatnia możliwość to przybycie do wierzchołka po raz pierwszy, wtedy także należy dorzucić go do otwartej listy.

Ostateczna efektywność algorytmu zależy od wyboru struktur danych reprezentujących obie listy, doboru funkcji heurystycznej oraz użytego grafu. Przy doborze grafu reprezentującego mapę rozgrywki, na szybkość działania algorytmu ma wpływ ilość wierzchołków znajdująca się w grafie oraz tzw. czynnik rozgałęzienia, czyli średnia ilość sąsiadów przypadająca na każdy z węzłów. Poniżej zostanie opisany wybór odpowiedniej struktury danych oraz funkcji heurystycznej.

3.1.2 Struktury danych

Podczas każdej iteracji algorytmu dodawane są oraz zdejmowane wierzchołki z otwartej i zamkniętej listy. Sprawdzana jest także aktualna przynależność wierzchołka do którejś z nich. Podczas operacji zdejmowania elementu z otwartej listy należy znaleźć element z najmniejszą wartością funkcji f.

Standardowym wyborem jest kolejka priorytetowa dla otwartej listy i tablica flag dla zamkniętej listy. Kolejka realizowana jest za pomocą kopca binarnego. Uporządkowanie drzewa, gwarantuje stały czas dostępu do najmniejszej wartości funkcji ewaluacji f, która znajduje się zawsze w korzeniu drzewa. Operacje wstawiania i usuwania z kolejki realizowane są w czasie $O(log(n))$, gdzie n to ilość wierzchołków znajdujących się w kolejce. Użycie tablicy flag pozwala na zaznaczanie i odznaczanie rozpatrzonych wierzchołków w czasie $O(1)$.

3.1.3 Funkcja heurystyczna

Algorytm A* różni się od Dijkstry priorytetyzacją wierzchołków w otwartej liście. Dla Dijkstry jest to dotychczasowy najmniejszy koszt podróży od punktu startowego s do wierzchołka n, oznaczony jako g, dla A* funkcja ewaluacji f, którą można zapisać jako:

$$f(n) = g(n) + h(n), \text{gdzie} \tag{3.1}$$

- n – aktualnie rozpatrywany wierzchołek
- f – funkcja ewaluacji
- g – dotychczasowy najmniejszy koszt podróży od punktu startowego s do wierzchołka n
- h – heurystyczne oszacowanie odległości od wierzchołka n do celu g

Efektywność A* i optymalność znalezionej ścieżki zależy od doboru heurystyki.

Heurystyka h jest dopuszczalna(ang. admissible) jeżeli dla każdego wierzchołka n zachodzi warunek $h(n) \leqslant \hat{h}(n)$, gdzie $\hat{h}(n)$ to prawdziwa odległość od wierzchołka n do celu g. Użycie dopuszczalnej heurystyki jest warunkiem gwarantującym optymalność algorytmu. [14] Modelowanie heurystyki pozwala na zmianę zachowania algorytmu. Niesie to kilka własności:

1. Jeżeli heurystyka h(n) jest dopuszczalna to algorytm zawsze zwróci optymalne rozwiązanie, jednak im mniejsza wartość h(n) tym większą przestrzeń wierzchołków trzeba przeszukać. W szczególności, gdy $h(n) = 0$, to A* zachowuje się identycznie jak Dijkstra
2. Jeżeli h(n) jest zawsze równa $\hat{h}$(n) to rozpatrywane będą jedynie wierzchołki należące do optymalnej ściezki. Jest to sytuacja idealna, do której dąży się przy wyborze funkcji h.

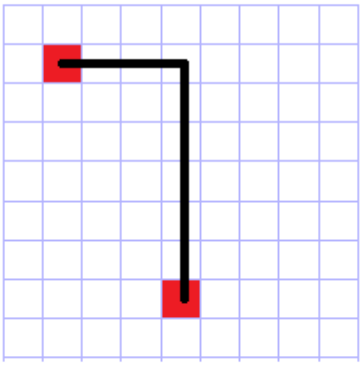
3. Jeżeli h(n) jest czasami większa od $\hat{h}(n)$, funkcja h(n) jest wtedy bardziej priorytezowana niż g(n), co powoduje wcześniejsze rozpatrywanie wierzchołków znajdujących się bliżej celu, nawet jeśli koszt podróży do nich jest większy. To z kolei zmniejsza ogólną ilość rozpatrywanych wierzchołków przez co algorytm działa szybciej. W ten sposób znaleziona ścieżka może być dłuższa od optymalnej. Jakość znalezionego rozwiązania zmniejsza się wraz z coraz większym przeszacowywaniem funkcji h(n).

3.1.4 Rozstrzyganie remisów

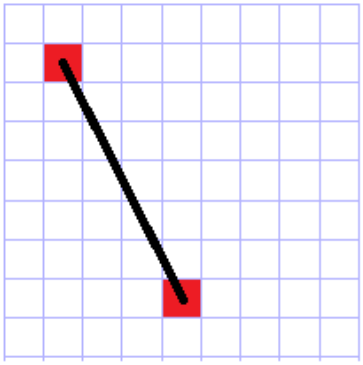
Częstym zjawiskiem jest posiadanie przez dwa lub więcej wierzchołków tej samej f-wartości pomimo różnych kosztów g i h. Podczas działania A*, każdy z nich jest rozpatrywany, co powoduje zwiększoną przestrzeń poszukiwań, przez co czas działania algorytmu wydłuża się. Można zastosować wtedy metodę rozstrzygania remisów, w celu większego faworyzowania jednej z wartości, g lub h. Przykładowy sposób radzenia sobie z tą sytuacją to zwiększenie wartości h. Spowoduje to, że dotychczas te same f-wartości będą się różnić i wierzchołki bliżej celu będą rozpatrywane wcześniej. Używając tej strategii rozwiązywania remisów heurystyka przestaje być dopuszczalna. [5]

3.1.5 Przykładowe heurystyki

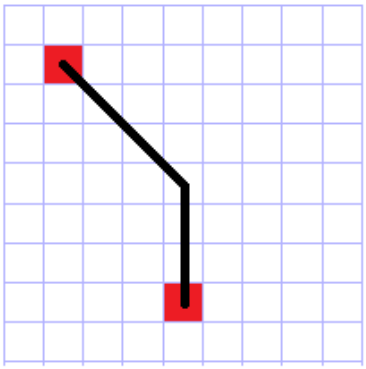
Dobór odpowiedniej heurystyki w dużej mierze zależy od użytego grafu reprezentującego mapę rozgrywki. Dla map kratowych można wyróżnić trzy najczęściej stosowane. W poniższych przykładach punkt startowy znajduje się w punkcie (x_1, y_1), natomiast cel w punkcie (x_2, y_2). Poprzez stałą *dist* rozumie się najmniejszą odległość między dowolnymi sąsiednimi wierzchołkami na mapie. Wybór takiej wartości zagwarantuje dopuszczalność heurystyki.



Rysunek 3.1: Odległość Manhattan



Rysunek 3.2: Odległość Euklidesowa



Rysunek 3.3: Odległość ukośna

Odległość Manhattan (ang. Manhattan distance)

Funkcję można wyrazić w następujący sposób:

$$h = dist \cdot (|x_1 - x_2| + |y_1 - y_2|) \tag{3.2}$$

Zwrócone zostaje oszacowanie odległości między wierzchołkami, które uwzględnia możliwość ruchu jedynie w 4 strony. Dla map umożliwiających ruch w większą ilość stron, heurystyka zawyża rzeczywistą odległość, przez co przestaje być dopuszczalna. Plusem użycia tej heurystyki w wypadku bycia niedopuszczalną, jest jej prostota, zapewniająca szybkie obliczenia. Przedstawiona została na rysunku 3.1.

Odległość Euklidesowa (ang. Euclidean distance)

Funkcję można wyrazić w następujący sposób:

$$h = dist \cdot \sqrt{d_x \cdot d_x + d_y \cdot d_y}, \tag{3.3}$$

$$\text{gdzie } d_x = |x_1 - x_2|,\ d_y = |y_1 - y_2|$$

Heurystyka zwraca odległość między dwoma punktami, która byłaby pokonana poruszając się bezpośrednio w kierunku celu. Heurystyka zwraca niedoszacowaną wartość, bądź równą rzeczywistej, jest więc dopuszczalna. Używanie tej heurystyki może skutkować problemami. Jest ona kosztowna obliczeniowo z uwagi na dwa działania mnożenia i jedno pierwiastkowania. Drugim problemem jest częste niedoszacowanie wartości, które skutkuje rozpatrywaniem wielu niepotrzebnych wierzchołków, szczególnie dla map kratowych z brakiem możliwości ruchu w każdym kierunku. Przedstawiona została na rysunku 3.2.

Odległość ukośna (ang. Diagonal distance)

$$h = dist \cdot max(d_x, d_y) + (diag - 2 \cdot dist) \cdot min(d_x, d_y), \tag{3.4}$$

$$\text{gdzie } d_x = |x_1 - x_2|,\ d_y = |y_1 - y_2|$$

Dla $dist = 1$ i $diag = \sqrt{2}$ funkcja przyjmuje postać:

$$h = max(d_x, d_y) + (\sqrt{2} - 2) \cdot min(d_x, d_y), \tag{3.5}$$

Stała `dist` jest tutaj najmniejszą odległością miedzy sąsiednimi wierzchołkami nie uwzględniając ruchu po ukosie, `diag` natomiast najmniejszą odległością między sąsiednimi wierzchołkami po ukosie. Heurystyka nadaje się idealnie do map w kratę z możliwością ruchu w 8 stron. Przedstawiona została na rysunku 3.3.

3.1.6 Odzyskiwanie ścieżki

Dzięki zapisywaniu rodzica każdego wierzchołka na ścieżce, rekonstrukcję ścieżki można zobrazować prostym pseudokodem.

Pseudokod 3.2: Rekonstrukcja ścieżki

```
1 CurrNode = GoalNode;
2 while parent[CurrNode]! = null do
3 |   add CurrNode to beginning of found path;
4 |   CurrNode = parent[CurrNode];
5 add StartNode to beginning of found path
```

Dopóki rozpatrywany wierzchołek ma przypisanego rodzica, zostaje on dodany do finalnej ścieżki, następnie rozpatrywany jest rodzic tego wierzchołka.

3.1.7 Złożność czasowa

Złożoność czasowa zależy od wyboru heurystyki. Niech:

- N – ilość rozpatrzonych wierzchołków,
- d – długość najkrótszej ścieżki,
- b – branching factor,
- b* – efektywny branching factor – średnia ilość sąsiadów rozpatrywana dla każdego wierzchołka, wtedy górne ograniczenie na koszt czasowy algorytmu wynosi:

$$N = b^* + (b^*)^2 + ... + (b^*)^d, \tag{3.6}$$

czyli $O(b^d)$. Ilość rozpatrywanych wierzchołków jest eksponencjalnie większa od długości najkrótszej ścieżki.

W przypadku, gdy rozpatrywana przestrzeń wyszukiwania jest drzewem i funkcja h spełnia poniższy warunek: $|h(x) - h^*(x)| = O(log h^*(x))$, gdzie h* jest optymalną heurystyką [13], zwracającą zawsze dokładny koszt od aktualnie rozpatrywanego wierzchołka do celu, czas działania algorytmu jest wielomianowy i wynosi O(d). Warunek oznacza, że błąd funkcji h nie rośnie szybciej, niż logarytm z funkcji h*. Rozpatrywane są jedynie wierzchołki należące do najkrótszej ścieżki.

3.1.8 Złożoność pamięciowa

Dla każdego rozpatrywanego wierzchołka zapamiętywana jest odległość g, oszacowanie h, jego rodzic oraz flaga informująca czy jest on w zamkniętej liście. Zakładając, że na przechowanie tych wartości przeznaczone zostanie 16 bajtów, to dla dużych map rzędu 1024×1024 zużycie pamięci nie przekroczy 16MB. Jest to mocno zawyżona wartość z uwagi na użycie funkcji heurystycznej, która mocno zmniejszy prawdziwe zużycie pamięci. Dla nowoczesnego sprzętu zarówno na PC, jak i konsoli nie jest to problem. [10]

3.2 Algorytm Hierarchical Pathfinding A* (HPA*)

Algorytmy hierarchiczne przyjmują inną strategię niż A*. Zamiast wykonywać jedno długie wyszukiwanie, problem dzielony jest na kilka serii mniejszych wyszukiwań. Ideą algorytmu jest podział kosztownego problemu na wiele o niskim koszcie. Jest analogiczny do sposobu planowania trasy przez ludzi. Chcąc przemieścić się do galerii handlowej po drugiej stronie miasta, może zajść potrzeba jazdy dwoma tramwajami. Najpierw znajdowany jest przystanek, z którego rozpoczyna się jazda tramwajem, następnie kolejny, w którym zachodzi potrzeba przesiadki. W celu dostania się na pierwszy przystanek trzeba najpierw dojść do wyjścia z mieszkania, potem wyjścia z klatki schodowej, następnie przejść przez jezdnię w odpowiednim miejscu i tak dalej. Główne poszukiwanie zostało zastąpione wieloma mniejszymi, trywialnymi w porównaniu do początkowej skali problemu.

Hierarchiczny algorytm planuje najpierw ścieżkę na najwyższym poziomie abstrakcji, żeby potem stopniowo zamieniać ją na bardziej szczegółową na niższych poziomach abstrakcji. Finalnie wyznaczana jest ostateczna trasa na najniższym poziomie. Plusem takiego rozwiązania jest możliwość uszczegóławiania ścieżki jedynie, jeśli zajdzie taka potrzeba. W przypadku poruszania się bohatera z jednego końca mapy na drugi, może istnieć spora szansa, że w pewnym momencie droga na wyznaczonej ścieżce zostanie zablokowania. Należy wtedy wyznaczyć nową, nie jest to kłopotliwe z uwagi na to, że tylko niewielki czas na wyszukiwanie poprzedniej został zmarnowany. Kolejnym plusem rozpatrywania jedynie lokalnego otoczenia podczas uszczegóławiania jest krótki czas, po którym bohater może zacząć poruszać się w wyznaczonym kierunku, jednocześnie mając pewność, że porusza się on w dobrą stronę. W A*, aby zacząć podróż do celu należałoby najpierw zaczekać, aż zostanie wyznaczona kompletna ścieżka.

Algorytm składa się z dwóch głównych etapów, którymi są:

1. faza preprocessingu
2. wyszukiwanie ścieżki

3.2.1 Preprocessing

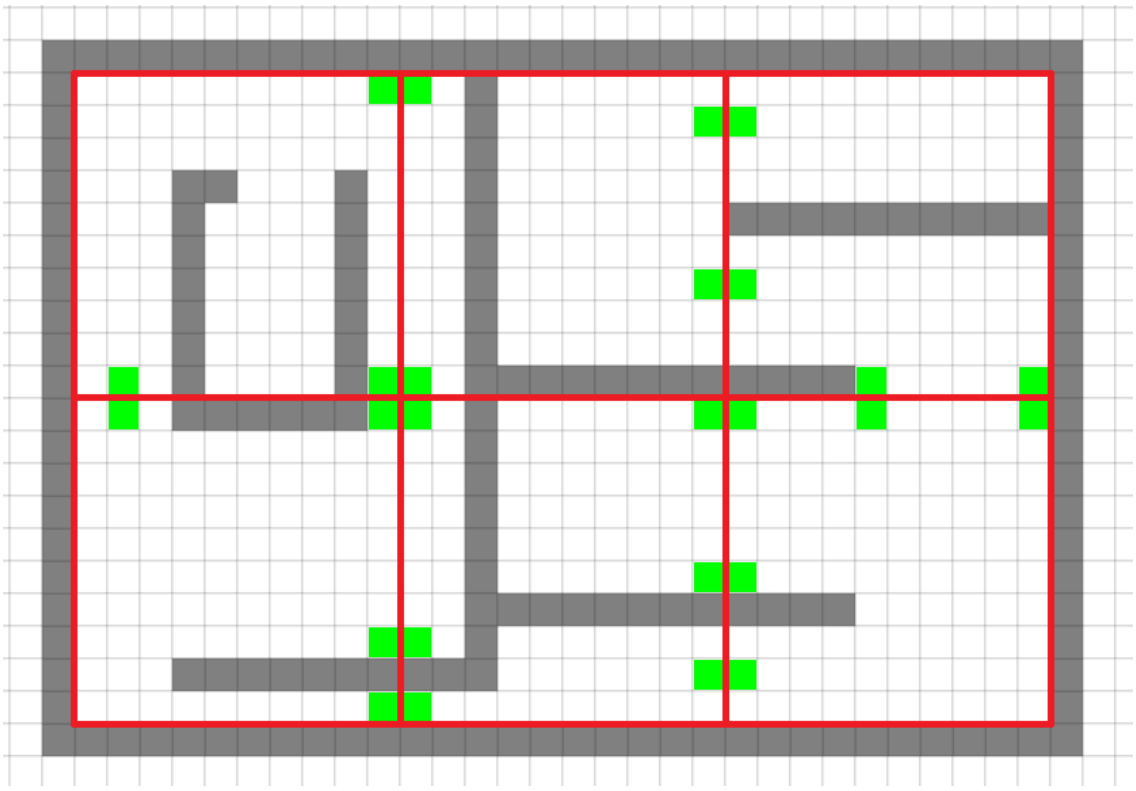
Etap ten polega na stworzeniu grafu odpowiadającego każdemu dodanemu poziomowi abstrakcji. Obrazuje go pseudokod 3.3.

Pseudokod 3.3: Preprocessing w HPA*

```
1  Function preprocessing(maxLevel) {
2  buildEntrances()
3  buildGraph(1)
4  for lvl = 2 to maxLevel do
5      buildGraph(lvl)
6  }
7
8  Function buildEntrances(){
9  C[1] = buildClusters(1)
10 foreach c1,c2 ∈ C[1] do
11     if adjacent(c1,c2) then
12         Entrances = Entrances ∪ createEntrance(c1, c2)
13 }
14
15 Function buildGraph(level){
16 foreach c1,c2 ∈ C[level] do
17     if adjacent(c1,c2) == false then
18         continue
19     foreach adjacent e1 ∈ c1 and e2 ∈ c2 do
20         n1 = newNode(e1, c1)
21         n2 = newNode(e2, c2)
22         addNode(n1)
23         addNode(n2)
24         addEdge(n1, n2, INTER)
25 foreach c ∈ C[1] do
26     foreach n1, n2 ∈ Nodes[c], n1 ≠ n2 do
27         dist = searchForDistance(n1, n2, c)
28         if d < ∞ then
29             addEdge(n1, n2, d, INTRA)
30 }
```

Na każdym poziomie abstrakcji mapę dzieli się na mniejsze prostokątne obszary zwane klastrami, następnie dla każdego z klastrów tworzone są wejścia. Wejście jest linią znajdującą się na obrzeżu klastra, na której wyznaczane są punkty tranzytowe. Autorzy algorytmu proponują wybranie jednego takiego punktu na środku wejścia, w przypadku długości wejścia poniżej ustalonej wartości, oraz dwóch takich punktów na krańcach wejścia, w przeciwnym przypadku. [11] Punktem tranzytowym nazywany jest wierzchołek znajdujący się na wejściu klastra, posiadający sąsiadujący do niego wierzchołek na wejściu sąsiadującego klastra. Następnie oba wierzchołki są łączone i stworzona krawędź dodawana jest do grafu. Krawedź dostaje miano krawędzi międzyklastrowej(ang. INTER edge).

Następnie znajdowane są wszystkie odległości między punktami tranzytowymi w każdym z klastów. Odległości odpowiadają wagom krawędzi między tymi wierzchołkami. Taka krawedź dodawana jest do grafu tworzącego dany poziom i nazywana jest krawędzią wewnątrzklastrową (ang INTRA edge). W analogiczny sposób dodawane są kolejne poziomy abstrakcji i odpowiadające im grafy. Jedyna różnicą jest wyszukiwanie punktów tranzytowych na wyższych poziomach abstrakcji. Brane pod uwagę są jedynie wcześniej już stworzone punkty tranzytowe, aby w sytuacji uszczegóławiania ścieżki i obniżenia poziomu rozpatrywanego grafu, aktualny punkt tranzytowy dalej znajdował się w tym samym miejscu. Przykład podziału mapy na klastry oraz wyznaczenia wejść tranzytowych przedstawiony został na rys. 3.4, natomiast na rys 3.5 zaznaczone zostały krawędzie wewnątrz i międzyklastrowe.



Rysunek 3.4: Przykładowa mapa z podziałem na klastry(czerwone linie) oraz punkty tranzytowe(zielone kraty)

Zasady wyboru wejść

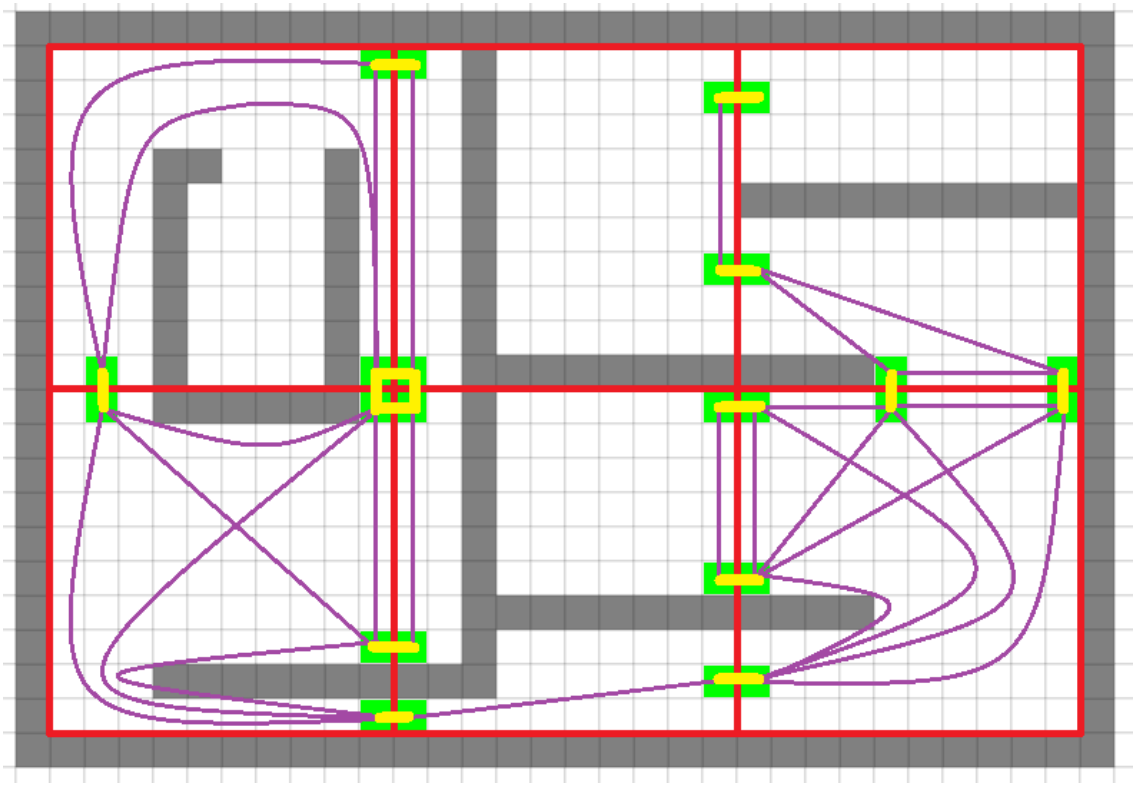
Istnieje kilka zasad, które musi spełnić każde wejście:

1. Warunek ograniczenia granicy: $e \subseteq l_1 \cup l_2$, gdzie e – wejście do klastra, l_1, l_2 - dwie sąsiednie linie płytek w każdym z klastrów, określające granice między sąsiednimi klastrami. Oznacza on, że wejście e definiowane jest wzdłóż klastra i jego długość nie może przekroczyć długości granicy wzdłuż dwóch sąsiadujących klastrów.
2. Warunek symetrii: $\forall t \in l_1 \cup l_2 : t \in e \Leftrightarrow symm(t) \in e$, oznacza on, że dla każdego punktu tranzytowego t znajdującego się na granicy klastra musi istnieć odpowiadający mu punkt tranzytowy na granicy sąsiadującego klastra.
3. Warunek braku przeszkód: wejście nie może zawierać przeszkód.
4. Warunek maksymalnego spełnienia warunków: wejście rozszerzane jest w obie strony tak długo jak spełnione są powyższe warunki.

Uwagi

Faza preprocessingu jest najbardziej kosztowna czasowo. Może zostać ona wykonana na etapie produkcji gry, a następnie wczytana podczas ładowania konkretnego poziomu, albo wykonana bezpośrednio podczas ładowania poziomu.

W razie zmiany stanu możliwości ruchu przez dany wierzchołek, należy dokonać aktualizacji poziomów w abstrakcyjnym grafie. Trzeba ponownie wyznaczyć wagi krawędzi między punktami tranzytowymi w klastrze,



Rysunek 3.5: Połączenie krawędzi wewnatrzklastrowych (na fioletowo) oraz międzyklastrowych (na żółto)

w którym zaszła zmiana. W wypadku unieważnienia punktu tranzytowego między klastrami, należy wyznaczyć nowy, jeżeli takowy istnieje. Algorytm nadaje się więc do wykorzystania na mapach z dynamicznym środowiskiem.

3.2.2 Wyszukiwanie ścieżki

Proces wyszukiwania ścieżki można podzielić na kilka etapów:

1. dodanie punktu startowego i celu do grafu
2. wyznaczenie ściezki na najwyższym poziomie abstrakcji
3. redukcja wyznaczonej ścieżki do niższych poziomów
4. ewentualna poprawa jakości wyznaczonej ścieżki
5. usunięcie punktu startowego i celu z grafu

Obrazuje to pseudokod 3.4.

Pseudokod 3.4: Wyszukiwanie ścieżki w HPA*

```
1  Function HierarchicalSearch(start, goal, maxLevel) {
2  insertNode(start, maxLevel)
3  insertNode(goal, maxLevel)
4  abstractPath = searchPath(start, goal, maxLevel)
5  refinedPath = refinePath(abstractPath, maxLevel)
6  smoothedPath = smoothPath(refinedPath)
7  return smoothedPath
8  }
9
10 Function insertNode(node, maxLevel){
11 for lvl = 1 to maxLevel do
12 |   c = determineCluster(node, lvl)
13 |   connectToBorder(node, c)
14 }
15
16 Function connectToBorder(node, cluster){
17 foreach n ∈ Nodes[cluster] do
18 |   dist = searchForDistance(node, n, cluster)
19 |   addEdge(node, n, dist, INTRA)
20 }
21
```

Dodanie punktu startowego i celu do grafu

Aby móc poruszyć się z punktu startowego, musi zostać on włączony do grafu na każdym z poziomów abstrakcji. Odbywa się to poprzez ustalenie, w którym klastrze na danym poziomie leży wierzchołek i następnie wyznaczenie algorytmem A*, bądź jakimś innym, wag krawędzi do wszystkich punktów tranzytowych należących do klastra. Dodanie celu do grafu odbywa się analogicznie. Wraz z dodawaniem kolejnych poziomów abstrakcji oraz powiększaniem obszaru klastrów etap staje się coraz bardziej kosztowny, aż w pewnym momencie zaczyna dominować w całym procesie wyszukiwania. Należy więc uważnie dobrać powyższe parametry.

Wyznaczenie ściezki na najwyższym poziomie abstrakcji

Po tym etapie otrzymana jest najbardziej ogólna ścieżka, która w razie potrzeby może być dalej uszczegóławiana. Między dwoma wierzchołkami należącymi do tego samego klastra wyznaczana jest ścieżka na niższym poziomie za pomocą algorytmu A*.

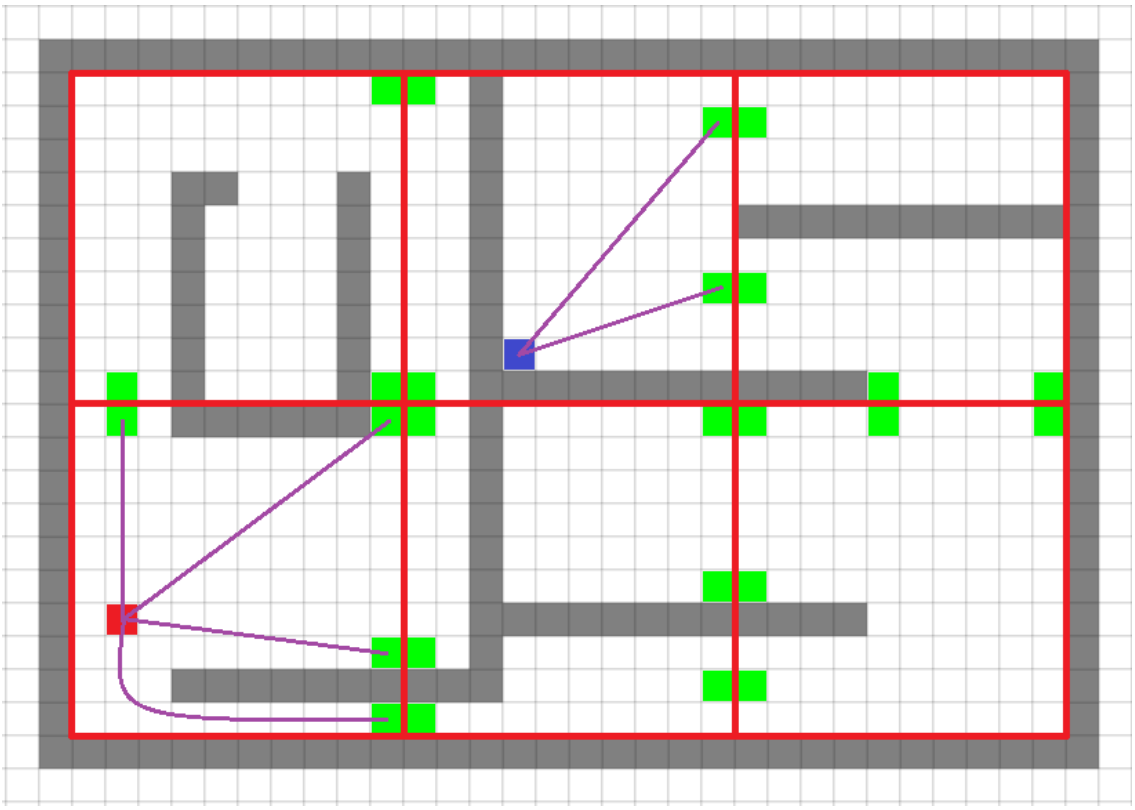
Ewentualna poprawa jakości wyznaczonej ścieżki

Z uwagi na stały wybór punktów tranzytowych otrzymana ścieżka na najniższym poziomie może nie być optymalna pod względem długości. Obrazuje to rysunek 3.7. W celu poprawy jej jakości można zastosować algorytmy wygładzające. Przykładowy algorytm może patrzeć, czy z jednego wierzchołka na wyznaczonej ścieżce można dostać się do kolejnych za pomocą prostej linii i jeśli jest to możliwe, to odpowiedni kawałek ścieżki zamieniany jest na nowo wyznaczony. [11]

Na koniec działania algorytmu należy usunąć dodane wcześniej punkty startowe i końcowe.

3.2.3 Złożoność czasowa

Przyspieszenie czasu działania HPA* względem A* wynika z podziału na wiele mniejszych wyszukiwań, które są o wiele mniej kosztowne w porównaniu do pojedyńczego wyszukiwania w A*. Podczas szukania na małym obszarze wewnątrz klastra, funkcja heurystyczna w A* lepiej się spisuje z uwagi na mniejszą liczbę przeszkód na swojej drodze, przez co rozpatrywana jest mniejsza liczba wierzchołków. W przypadku braku



Rysunek 3.6: Dodanie punktu startowego oraz celu do mapy, oraz połączenie ich z punktami tranzytowymi wewnątrz swoich klastrów

możliwości przejścia z jednej pozycji na mapie na drugą, HPA* wykryje taką sytuację podczas rozpatrywania grafu na najwyższym poziomie, natomiast A* rozpatrzy wszystkie wierzchołki do których możliwa jest podróż i dopiero wtedy zwróci informację o braku znalezienia ścieżki.

3.2.4 Złożoność pamięciowa

O ilości zużytej pamięci przez algorytm decyduje ilość dodanych poziomów abstrakcji, gdyż na każdym poziomie przechowywany jest odpowiadający mu graf. Według analizy autorów [11] pesymistyczna ilość krawędzi przechowywana na każdym z poziomów wynosi:

- dla krawędzi wewnątrzklastrowych:

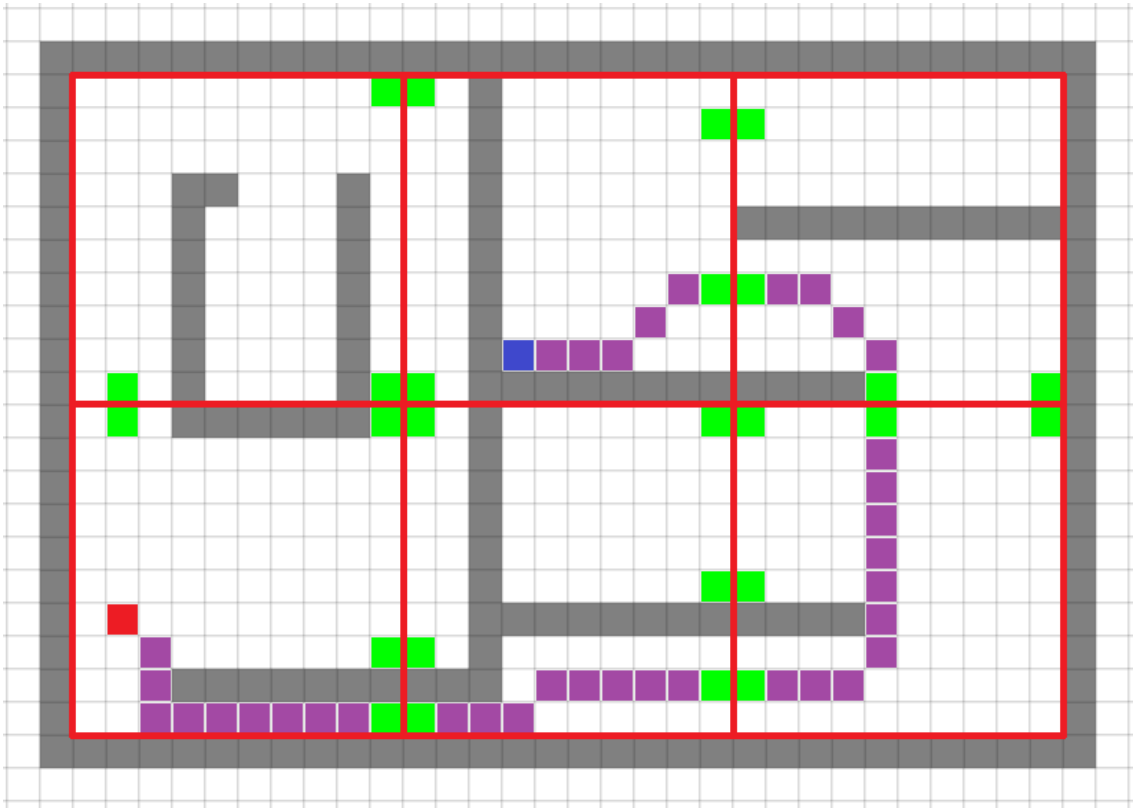
$$n(c-2)^2(2n-1) + 2(n-1) + 3(c-2)(1.5n-1)$$

- dla krawędzi międzyklastrowych:

$$m(c-1)$$

gdzie:

- $n \times n$ – rozmiar klastra,
- $m \times m$ – wielkość mapy,
- $c \times c$ – liczba klastrów na mapie

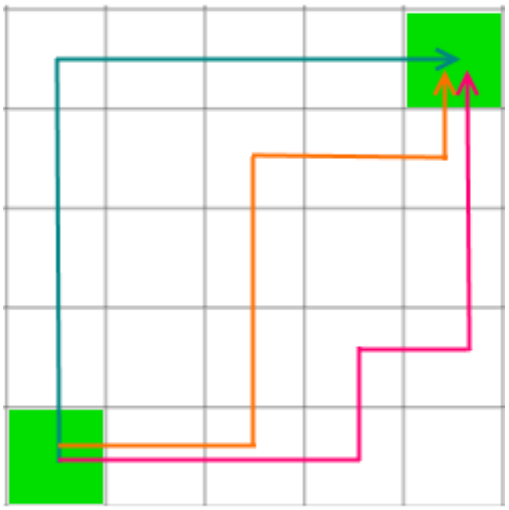


Rysunek 3.7: Wyznaczona ścieżka jest nieoptymalna, z uwagi na konieczność poruszania się między punktami tranzytowymi

Ilość zużytej pamięci podczas małych wyszukiwań algorytmem A* jest znacznie mniejsza względem pojedyńczego wyszukiwania tym samym algorytmem z uwagi na mniejsze obszary przeszukiwań i lepszy wybór wierzchołków przez funkcje heurystyczna.

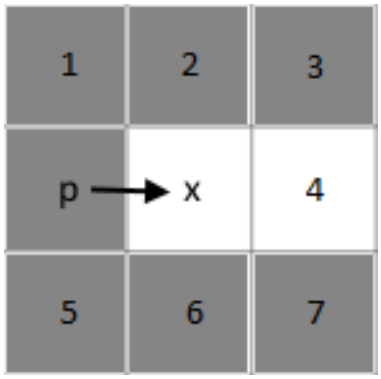
3.3 Jump Point Search (JPS)

JPS jest modyfikacją algorytmu A* przeznaczoną jedynie dla map na kracie, o jednolitym koszcie ruchu między płytkami, na przykład o koszcie 1 dla ruchu po prostej i $\sqrt{2}$ dla ruchu po przekątnej. Algorytm działa na zasadzie eliminacji symetrycznych ścieżek. Dwie ścieżki definiuje się jako symetryczne, jeśli mają wspólny punkt startowy i końcowy i jedną ścieżkę można otrzymać z drugiej poprzez zamianę kolejności wektorów kierunków, w jakich następuje ruch. Widać to na rysunku nr 3.8, gdzie każda ścieżka ma tę samą długość i zawiera pewną permutację 4 kroków w górę i 4 kroków w prawo. Algorytm A* w przypadku występowania symetrii rozpatruje wszystkie symetryczne ścieżki do celu. JPS stara się uniknąć tego problemu poprzez przeskakiwanie wielu wierzchołków, rozpatrując jedynie tak zwane punkty skoków (ang. jump points). Zostaną one przedstawione w dalszej części. W celu znalezienia punktu skoku algorytm stosuje zasady odcinania sąsiednich wierzchołków, w celu eliminacji tych wierzchołków, które nie muszą być rozpatrzone, aby dostać się optymalną ścieżką do celu.

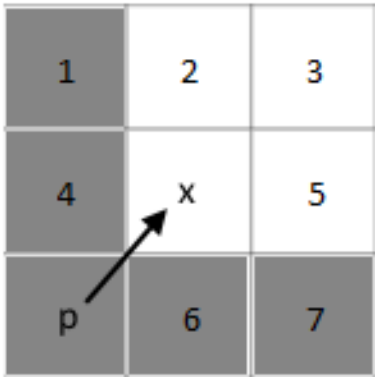


Rysunek 3.8: Przykładowe symetryczne ścieżki

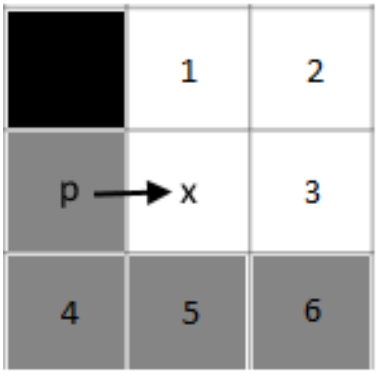
Istnieją trzy zasady odcinania sąsiadów:



Rysunek 3.9: Odcinanie sąsiadów podczas ruchu po prostej



Rysunek 3.10: Odcinanie sąsiadów podczas ruchu po skosie



Rysunek 3.11: Odcinanie sąsiadów podczas wystąpienia wierzchołka wymuszonego

1. Podczas ruchu po prostej od wierzchołka p, odcinane są wszystkie wierzchołki $n \in$ neighbours(x), dla których spełniony jest warunek dominacji:

$$len(\langle p(x), \ldots, n\rangle \backslash x) \leqslant len(\langle p(x), x, n\rangle), \tag{3.7}$$

gdzie:

- p(x) – rodzic x,
- $\langle \ldots \rangle$ oznacza ścieżkę, więc $\langle p(x), \ldots, n\rangle$ to ścieżka od p(x) do n, natomiast $\langle p(x), x, n\rangle$ ścieżka od p(x) przechodząca przez x do n,
- symbol \ oznacza wykluczenie wierzchołka ze ścieżki, więc $\langle p(x), \ldots, n\rangle \backslash x$ to ścieżka od p(x) do n, niezawierająca x,
- len – określa długość ścieżki.

Zgodnie z tą zasadą, na rysunku nr 3.9 można odciąć wszystkie szare wierzchołki, gdyż istnieje do nich inna optymalna ścieżka nieprzechodząca przez x. Na przykład do wierzchołka numer 3 można dostać się

tym samym kosztem pokonując ścieżkę $\langle p, 2, 3\rangle$ jak również $\langle p, x, 3\rangle$. Te dwie ścieżki różnią się jedynie permutacją wektorów kierunków.

2. Podczas ruchu po przekątnej obowiązuje analogiczny warunek jak powyżej, jednak teraz musi występować ostry warunek dominacji:

$$len(\langle p(x), \ldots, n\rangle \backslash x) < len(\langle p(x), x, n\rangle). \tag{3.8}$$

Sytuację przedstawiono na rys 3.10. Tu również wykluczone zostają wszystkie szare wierzchołki. Na przykład do wierzchołka 1 można dostać się mniej kosztowną ścieżką $\langle p, 4, 1\rangle$ nie przechodzącą przez wierzchołek x.

3. Podczas ruchu po prostej w przypadku wystąpienia przeszkody. Na rys 3.11 oznaczona została, jako czarna krata. Wprowadzone zostają pojęcia naturalnych sąsiadów i wymuszonych sąsiadów. Zakładając brak przeszkody dla sąsiadów x, wierzchołki, które nie zostaną odcięte nazywa się naturalnymi sąsiadami x. Wierzchołek nazywany jest wymuszonym jeśli:

   - nie jest naturalnym sąsiadem x,
   - zachodzi warunek:

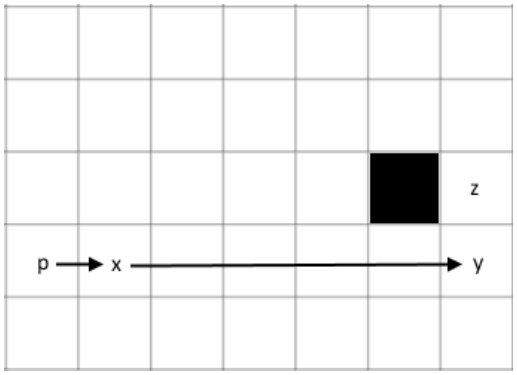
$$len(\langle p(x), x, n\rangle) < len(\langle p(x), \ldots, n\rangle \backslash x). \tag{3.9}$$

Na rysunku 3.11, wymuszonymi sąsiadami są wierzchołki numer 1 i 2. Nie można ich odciąć, gdyż nie istnieje do nich inna optymalna ścieżka, nie przechodząca przez x.

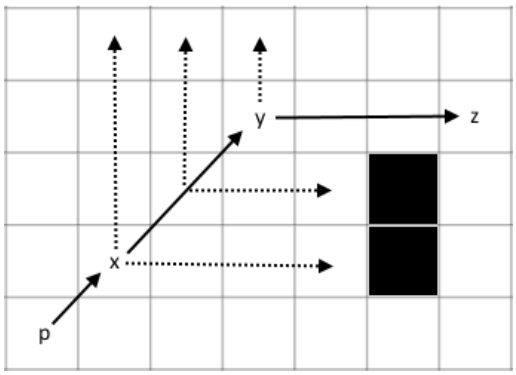
3.3.1 Punkty skoków

Wierzchołek y jest punktem skoku z wierzchołka x, poruszając się w kierunku d, jeśli y minimalizuje wartość k w taki sposób że $y = x + k \cdot d$ i zachodzi jeden z poniższych warunków:

1. Wierzchołek y jest celem.
2. Wierzchołek y ma przynajmniej jednego sąsiada który jest wymuszony.
3. Jeżeli d jest ruchem po przekątnej i istnieje wierzchołek $z = y + k_i \cdot d_i$, który leży $k_i \in N$ kroków w kierunku $d_i \in \{d_1, d_2\}$ tak, że wierzchołek z jest punktem skoku od wierzchołka y przez spełnienie warunku 1 lub 2, gdzie $\{d_1, d_2\}$ odpowiada kierunkowi pod kątem 45 stopni w stosunku do kierunku ruchu d.



Rysunek 3.12: Przykład skoku po prostej



Rysunek 3.13: Przykład skoku po ukosie

Na rysunku 3.12 ruch odbywa się po prostej i punktem skoku dla wierzchołka x jest wierzchołek y, ponieważ z jest wymuszony.

Poruszając się po przekątnej, w każdym kroku należy sprawdzić, czy nie został wykryty punkt skoku, w obu kierunkach pod kątem 45 stopni względem pierwotnego kierunku ruchu. Na rysunku 3.13 widać, że takim punktem jest z. Wierzchołek y jest więc punktem skoku dla wierzchołka x.

Pseudokod 3.5: Znajdowanie następników w JPS

```
1  Function identifySuccessors(x, start, goal) {
2  neighbours(x) = prune(x,neighbours(x))
3  foreach n ∈ neighbours(x) do
4  |  n = jump(x, direction(x, n), start, goal)
5  |  add n to successors(x)
6  return successors(x)
7  }
8
9  Function jump(x, direction, start, goal){
10 n = step(x,direction)
11 if n is an obstacle or is outside the grid then
12 |  return null
13 if n == goal then
14 |  return n
15 if ∃n' ∈ neighbours(n) which is forced then
16 |  return n
17 if direction is diagonal then
18 |  for i = 1 to 2 do
19 |  |  if jump(n, d_i, start, goal) is not null then
20 |  |  |  return n
21 return jump(n, direction, start, goal)
22 }
```

Opis algorytmu

Funkcja identifySuccessors(x) zwraca listę następników x, którą należy rozpatrzeć w analogiczny sposób jak listę sąsiadów wierzchołka x w algorytmie A*. Dla wierzchołka x, brane są pod uwagę naturalne i wymuszone wierzchołki. Dla każdego z nich następuje próba znalezienia punktu skoku i jeśli taki istnieje, dodawany jest on do listy następników x.

Funkcja jump działa w sposób rekurencyjny, próbując znaleźć punkty skoku. Badane są po kolei warunki z definicji punktu skoku. Jeżeli przy próbie wejścia na sąsiedni element w zadanym kierunku natrafi się na przeszkodę lub wyjdzie poza mapę, punkt skoku uznawany jest za nieznaleziony. W przypadku trafienia na cel zostaje on zwrócony, aby A* mógł uznać ścieżkę za znalezioną. Podczas napotkania wymuszonego sąsiada, musi on zostać rozpatrzony przez A*. Ostatni warunek odpowiada ruchowi po przekątnej. Należy sprawdzić, czy poruszając się w obu kierunkach pod kątem 45 stopni nie zostanie napotkany cel, bądź wymuszony wierzchołek. W takim wypadku wierzchołek n jest punktem skoku i musi zostać rozpatrzony. Jeżeli żaden z powyższych warunków nie zostanie spełniony następuje dalsze rekurencyjne przeszukiwanie w zadanym kierunku.

Koszt pamięciowy

JPS tak samo, jak A* nie wymaga etapu preprocessingu. Zużycie pamięci w JPS względem A* jest mniejsze z uwagi na dodawanie do otwartej listy jedynie wierzchołków, które są punktami skoków. Wierzchołki między aktualnie rozpatrywanym, a punktem skoku są ignorowane.

Koszt czasowy

Przetwarzanie poszczególnych wierzchołków podczas skoku odbywa się w czasie $O(1)$. Dodatkowo ignorowane są symetryczne ścieżki do sąsiadów aktualnie rozpatrywanego wierzchołka. Jedynie wybrani następnicy dodawani są do otwartej listy. Skutkuje to mniejszą liczbą operacji wstawiania i zdejmowania elementów

z kolejki priorytetowej, wykonywanych w czasie $O(log(n))$. Mniejsza liczba wierzchołków w otwartej liście również przyspiesza operacje na kolejce priorytetowej.

Jakość otrzymanej ścieżki

Funkcje odcinania wierzchołków oraz wyznaczania punktów skoków zachowują optymalność algorytmu. Dowód można znaleźć w [3].

O ostatecznej optymalności długości ścieżki zależy dobór funkcji heurystycznej h, która decyduje o kolejności wierzchołków zdejmowanych z otwartej listy. Jeżeli funckja h jest dopuszczalna, algorytm zawsze znajdzie najkrótszą ścieżkę, o ile taka istnieje.

3.4 Fringe Search (FS)

Fringe Search jest algorytmem próbującym zbalansować zużycie pamięci i szybkość działania algorytmów A* oraz Iterative Deepening A* (IDA*). Poniżej zostanie opisana metoda działania IDA*.

3.4.1 Iterative Deepening A*

IDA* jest rekurencyjnym algorytmem przeszukiwania w głąb. Jego główną zaletą jest niewielkie zużycie pamięci, którego górne ograniczenie wynosi $O(d)$, gdzie d to ilość wierzchołków na znalezionej ścieżce. W każdej iteracji algorytmu wykonywane jest przeszukiwanie w głąb, nie dalej niż do ustalonej granicy f. Początkowo jest to oszacowanie funkcji heurystycznej h. W przypadku braku znalezienia rozwiązania w danej iteracji, granica przesuwana jest na większą odległość i algorytm rozpoczyna przeszukiwanie od nowa. Z uwagi na zminimalizowanie wykorzystywanej pamięci, nie są przetrzymywane żadne dodatkowe informacje o odwiedzonych wierzchołkach. Z każdą iteracją algorytm zaczyna przeszukiwanie od punktu startowego powtarzając rozpatrzone wierzchołki z poprzedniej iteracji. W szczególności podczas ostatniej iteracji mogłaby zajść potrzeba rozpatrzenia wszystkich wierzchołków w grafie. Problem można rozwiązać poprzez zapamiętywanie wierzchołków znajdujących się na granicy poprzedniej iteracji i w kolejnej iteracji rozpoczynać wyszukiwanie od nich. Kolejnym problemem jest wielokrotne rozpatrywanie tych samych wierzchołków w przypadku wystepowania wielu ścieżek do celu, które nie muszą być nawet optymalne. Rozwiązaniem jest przeznaczenie dodatkowej pamięci na zaznaczanie wierzchołków już odwiedzonych i podczas odwiedzania wierzchołka sprawdzanie czy nie została znaleziona do niego inna, bardziej optymalna ścieżka. W przypadku użycia pamięci na przechowywanie dodatkowych informacji algorytm określa się mianem Memory Enhanced Iterative Deepening A* (ME-IDA*).

Pseudokod 3.6: Fringe Search

```
1  Insert start to F (Fringe)
2  g[start] = 0
3  parent[start] = null
4  f_limit = h(start)
5  found = false
6  while found = false and F not empty do
7      f_min = ∞
8      onlyLowerFVal = true
9      foreach n ∈ F do
10         f = g(n) + h(n)
11         if f > f_limit then
12             f_min = min(f, f_min)
13             continue
14         if n == goal then
15             found = true
16             break
17         foreach s ∈ successors(n) do
18             g_s = g(n) + cost(n, s)
19             if g_s ⩾ g(s) then
20                 continue
21             if F contains s then
22                 remove s from F
23             Insert s into F after n
24             g(s) = g_s
25             parent[s] = n
26         Remove n from F
27     if for all nodes in F their f-val is higher than current f_limit then
28         f_limit = f_min
```

Opis algorytmu

Fringe F jest odpowiednikiem otwartej listy z A*, jednak elementy należące do F nie są uporządkowane, jak ma to miejsce w A*. F przechowuje wierzchołki, które należy rozpatrzeć i może zostać zaimplementowana jako lista dwukierunkowa. Analogicznie do IDA*, FS w każdej iteracji rozpatruje jedynie wierzchołki, których f-wartość jest mniejsza od f_{limit} – granicy poszukiwania w zadanej iteracji. Dla każdej iteracji głównej pętli, z uwagi na brak uporządkowania elementów w F, należy przejrzeć wszystkie elementy znajdujące się w F. Również podczas tej iteracji następuje wybór nowej granicy, dla przyszłej iteracji. Nowa granica jest najmniejszą f-wartością spośród rozpatrzonych wierzchołków w danej iteracji, ale jednocześnie większa niż aktualna granica. W celu zachowania optymalności granicę można przesunąć tylko, jeśli na liście F nie znajduje się wierzchołek z f-wartością mniejszą lub równą dotychczasowej granicy. Następnie rozpatrywany jest każdy z następników aktualnego wierzchołka. Jeśli można dostać się do któregoś z nich mniejszym kosztem niż dotychczas, sprawdzane jest czy znajduje się już w F liście i jeśli tak zostaje on z niej usunięty i ponownie dodawany z nową f-wartością. Po rozpatrzeniu wierzchołka z F listy, jest on z niej usuwany. Na koniec głównej iteracji, w razie potrzeby ustalana jest nowa granica.

3.4.2 Porównanie względem A*

1. Fringe używa listy dwukierunkowej zamiast kolejki priorytetowej, co daje stały czas operacji wstawiania i usuwania elementów.

2. Z uwagi na brak uporządkowania elementów w liście następuje częste rozpatrywanie tych samych wierzchołków, jednak czas tej operacji jest stały.

3. Uporządkowanie elementów w A* pozwala na szybsze znalezienie celu.

4. Mniejsze zużycie pamięci względem A*, ponieważ podczas uaktualniania f-wartości konkretnego wierzchołka, jest on najpierw usuwany z listy, a potem ponownie wstawiany. W A* nie są wykonywane analogiczne czynności, więc znajdują się tam poprzednie wyliczone f-wartości dla każdego z wierzchołków, o ile nie zostały wcześniej rozpatrzone i usunięte.

O finałowym rezultacie, który z dwóch algorytmów będzie szybszy, decydują trzy pierwsze powyższe czynniki.

Wyniki i porównanie

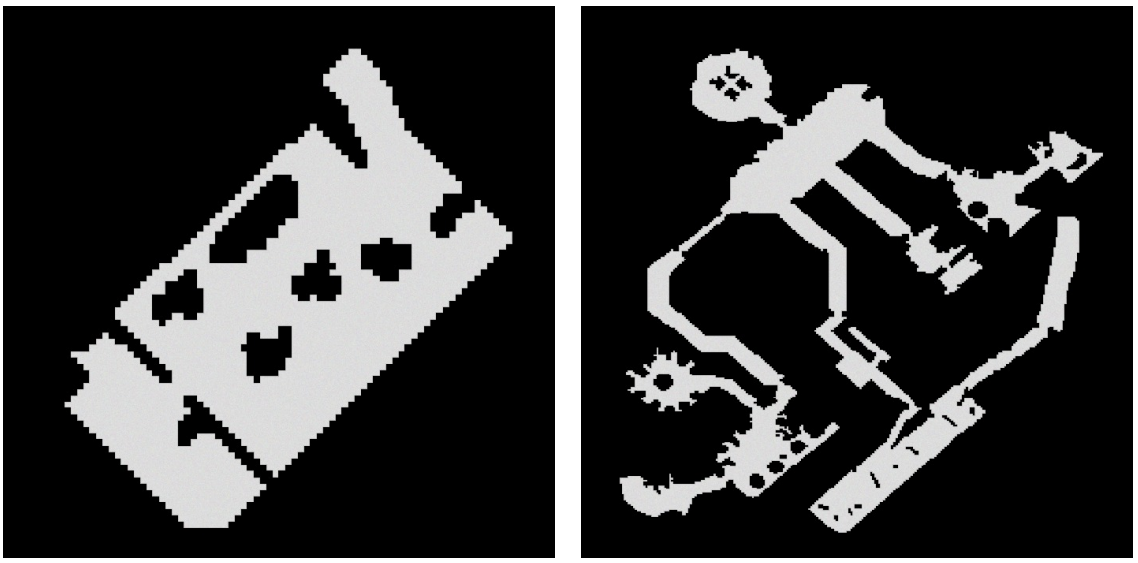
W tym rozdziale przedstawione zostaną wyniki przeprowadzonych eksperymentów, oraz ich interpretacja. Eksperymenty przeprowadzono na 3 zbiorach map w kratę ośmiokątną, umożliwiającą ruch w 8 stron, 4 po prostej oraz 4 po ukosie. Koszt poruszania się po prostej ustalony został na stałą wartość 1, a po przekątnej na 1.4142. Zbiory zostały pobrane z publicznego repozytorium, dostępnego online, upublicznionego przez [15]. Dla każdej z map wykonano po 400 wyszukiwań, dla każdego algorytmu. Punkty startowe oraz końcowe zostały wygenerowane przed właściwym procesem wyszukiwania i każdy z algorytmów został wykonany dla tych samych danych wejściowych co inne. Punkty startowe i końcowe zostały dobrane tak, aby nie pokrywały się oraz istniała między nimi możliwa do pokonania ścieżka. Aby zagwarantować drugi z powyższych warunków, przed zaakceptowaniem punktu i dodaniem go do zbioru testowego wykonano próbne wyszukiwanie algorytmem A*. Dopuszczenie do zbioru testowego pary punktów, między którymi niemożliwe jest wyznaczenie ścieżki skutkowałoby rozpatrzeniem wszystkich wierzchołków, do których możliwe jest dotarcie od punktu startowego. Wyjątkiem jest algorytm HPA*, dla którego zostałoby to stwierdzone już w pierwszym wyszukiwaniu na najwyższym poziomie abstrakcji. Istniałyby wtedy spore zaburzenia w przeprowadzonych pomiarach. Mapy są bardzo zróżnicowane między sobą. Dwie przykładowe mapy z Baldur's Gate II można zobaczyć na rysunku 4.1. Podczas wykonywania eksperymentów wzięty pod uwagę został wpływ użytej heurystyki, długość znalezionej ścieżki, czas wyszukiwania w ms oraz odchylenie standardowe próby dla czasu wyszukiwania, ilość rozpatrzonych i odwiedzonych wierzchołków, oraz średnia największa ilość elementów znajdujących się w otwartej liście na każdej z map. Dane zostały uśrednione dla każdego wyszukiwania. Oznaczenie widoczne przy wybranej heurystyce octile * wartość oznacza, że użyta została heurystyka ukośna zawyżająca oszacowanie o czynnik 'wartość'. Dzięki temu wierzchołki bliżej celu rozpatrywane są z większym priorytetem, niż te bliżej startu. Podczas eksperymentów, dla algorytmu HPA* został dodany tylko jeden dodatkowy poziom abstrakcji. Klastry są stałego rozmiaru 10×10 płytek. Każde wejście do klastra, o długości mniej niż 6 zawiera 1 punkt tranzytowy pośrodku wejścia, powyżej tej wielkości, dodawane są dwa punkty tranzytowe, po przeciwnych stronach wejścia. Parametry zostały zaproponowane przez autorów algorytmu i w średnim przypadku sprawdzają się najlepiej. Dodanie dodatkowego poziomu abstrakcji przynosi zysk w rzadkich przypadkach, a dobór odpowiednich parametrów nie jest łatwym zadaniem.

4.1 Wpływ heurystyki

W tym podrozdziale przedstawiony zostanie wpływ działania algorytmów w zależności od zastosowanej heurystyki. Pomiary dokonane zostały na zbiorze zawierającym 75 map z gry Baldur's Gate II, przeskalowanych do stałej rozdzielczości 512×512 płytek.

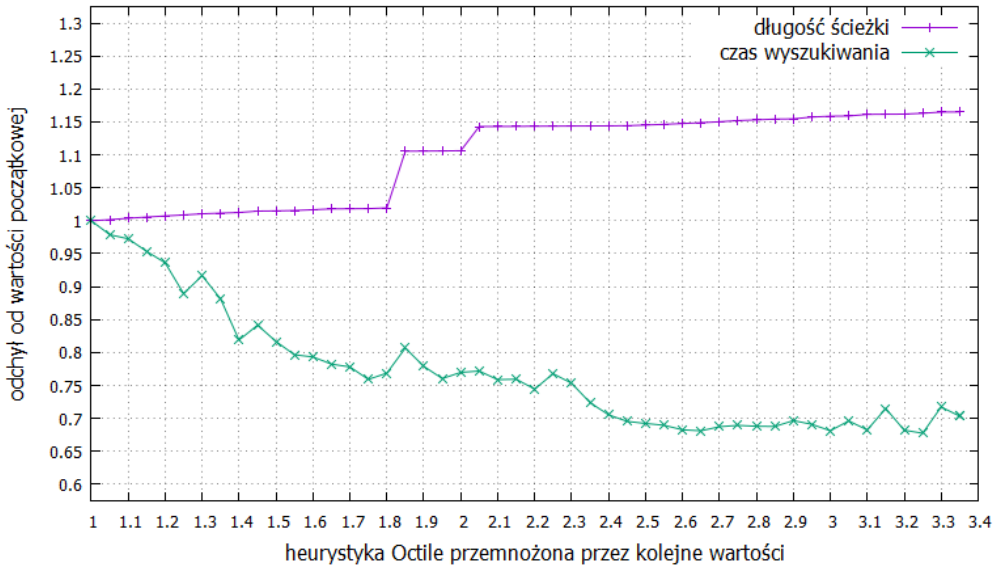
Na podstawie algorytmu A*, zostanie przedstawiony wpływ zawyżania wartości zwróconej przez heurystykę ukośną. Przedstawione zostało to na rysunku 4.2. Na osi x, znajdują się kolejne wartości, przez które przemnożona została wartość zwrócona przez heurystykę ukośną, zaś na osi y, stosunek otrzymanej długości ścieżki, bądź czasu wyszukiwania algorytmu, przy użyciu zmienionej heurystyki, do podstawowej heurystyki. Poprzez użycie zawyżonego oszacowania, funkcja heurystyczna przestaje być dopuszczalna. Znaleziona ścieżka przestaje być optymalna i jej długość zwiększa się, wraz ze wzrostem oszacowania. W rozpatrywanym przypadku, aż do wartości x równej 1.8, wzrost ten był niewielki. Jednak dla kolejnej wartości można zaobserwować nagłe, około 9% pogorszenie kosztu ścieżki. Świadczy to o nieprzewidywalności uzyskiwanych rezultatów dla nawet lekkich zmian oszacowania. Wraz z agresywniejszym oszacowaniem heurystyki, czas działania algorytmu znacznie maleje. Przykładowo dla wartości x równej 1.75, przy zachowaniu kilkuprocentowego odchyłu długości ścieżki, czas wyszukiwania zmalał o około 24%. Możliwe jest dalsze przyspieszenie algorytmu za cenę

jakości znalezionego rozwiązania. Właściwy dobór oszacowania pozwala na uzyskanie widocznie szybszego czasu działania algorytmu, przy jednoczesnym niewielkim pogorszeniu kosztu znalezionego rozwiązania.



(a) (b)

Rysunek 4.1: Przykładowe mapy z gry Baldur's Gate II



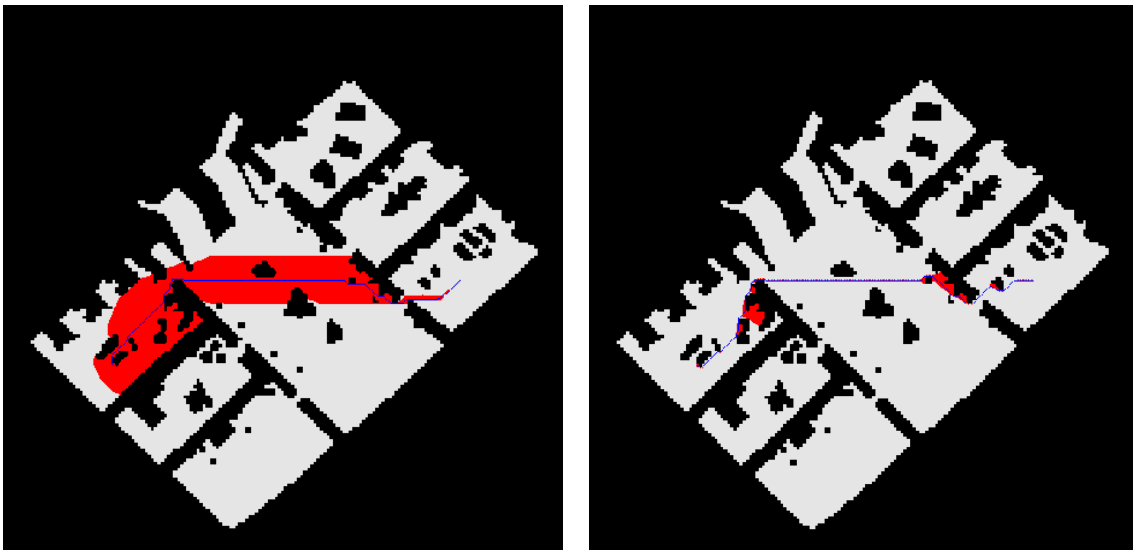
Rysunek 4.2: Wpływ zawyżania oszacowania funkcji heurystycznej na działanie algorytmu A*

4.1.1 Algorytm A*

heurystyka	długość znalezionej ścieżki	czas wyszukiwania w ms	st. odchylenie średniego czasu wyszukiwania	rozpatrzone wierzchołki	odwiedzone wierzchołki	największa ilość elementów OPEN listy
Octile	201.157	2.26518	1.20786	5631.55	8801.68	1092.13
Octile * 1.75	204.862	1.91146	1.08861	3067.57	5914.74	547.416
Manhattan	201.473	1.95694	1.05267	4378.41	7630.91	471.529
Euklidesowa	201.157	2.86589	1.30073	7134.63	10022.7	551.441

Tablica 4.1: Wyniki wyszukiwań algorytmem A* przy zastosowaniu różnych heurystyk

Agresywniejszy dobór heurystyki pozwala na przyspieszenie działania algorytmu A*. Objawia się to zmniejszoną ilością rozpatrywanych wierzchołków, oraz mniejszą ilością elementów w otwartej liście. Algorytm rozpatruje mniejszą ilość sąsiadów dla każdego z wierzchołków, z uwagi na priorytezację tych znajdujących się bliżej celu. Różnicę przedstawiono odpowiednio na rys 4.3.a oraz 4.3.b. Jest to jednak obarczone znalezieniem nieoptymalnej ścieżki. Heurystyka Euklidesowa nie jest dopasowana do reprezentacji grafu. Rozpatruje ona ruch w każdym kierunku, bezpośrednio do celu, tymczasem mapa pozwala na ruch jedynie w 8 stron. Skutkuje to zwiększeniem wierzchołków, które należy rozpatrzyć. Czas działania algorytmu jest to przez to znacznie gorszy niż w innych przypadkach. Heurystyka Manhattan w porównaniu do octile*1.75 działa w prawie identycznym czasie, przy jednoczesnej dużo większej ilości rozpatrywanych i odwiedzonych wierzchołków. Powodami są mniejsza ilość elementów w otwartej liście oraz mniej kosztowne obliczeniowo wyznaczanie oszacowania.



(a) (b)

Rysunek 4.3: Na czerwono zaznaczona została ilość rozpatrzonych wierzchołków dla heurystyki a) ukośnej b) ukośnej · 1.75

4.1.2 HPA*

heurystyka	długość znalezionej ścieżki	czas wyszukiwania w ms	st. odchylenie średniego czasu wyszukiwania	rozpatrzone wierzchołki	odwiedzone wierzchołki	największa ilość elementów OPEN listy
Octile	233.215	0.410053	0.173164	583.453	1259.46	110.643
Octile * 1.75	233.224	0.433407	0.168225	575.486	1257.31	110.583
Manhattan	233.215	0.389639	0.161782	548.889	1179.46	110.361
Euklidesowa	233.215	0.408487	0.177023	581.35	1222.44	110.648

Tablica 4.2: Wyniki wyszukiwań algorytmem HPA* przy zastosowaniu różnych heurystyk

Z uwagi na podział problemu wyszukiwania w HPA* na wiele mniejszych i mniej kosztownych podproblemów, dobór heurystyki ma tutaj znikome znaczenie i nie wpływa znacząco na żaden z rozpatrywanych parametrów, nawet dla niedopasowanej do grafu heurystyki Euklidesowej. Różnica między dopuszczalną heurystyką ukośną, a Manhattan to różnica około 5%.

4.1.3 Jump Point Search

heurystyka	długość znalezionej ścieżki	czas wyszukiwania w ms	st. odchylenie średniego czasu wyszukiwania	rozpatrzone wierzchołki	odwiedzone wierzchołki	największa ilość elementów OPEN listy
Octile	201.157	0.996335	0.585225	43.2209	20691.8	10.0151
Octile * 1.75	203.025	0.801874	0.508974	34.7655	18545.1	9.5569
Manhattan	201.469	0.814886	0.50868	39.3553	19356.1	9.70143
Euklidesowa	201.157	0.985829	0.564874	47.3582	22612.3	10.571

Tablica 4.3: Wyniki wyszukiwań algorytmem Jump Point Search przy zastosowaniu różnych heurystyk

Zastosowanie agresywniejszego oszacowania ukośnego, daje w tym przypadku około 20% przyspieszenie algorytmu, dzięki wcześniejszemu wybieraniu punktów skoków znajdujących się bliżej celu.

4.1.4 Fringe Search

heurystyka	długość znalezionej ścieżki	czas wyszukiwania w ms	st. odchylenie średniego czasu wyszukiwania	rozpatrzone wierzchołki	odwiedzone wierzchołki	największa ilość elementów OPEN listy
Octile	201.157	10.4181	10.4929	5567.31	699059	269.504
Octile * 1.75	221.671	52.2209	98.3064	17848	3417930	345.105
Manhattan	210.771	7.76984	7.42448	8919.2	351974	287.281
Euklidesowa	201.157	26.0277	17.2537	7151.05	1769280	306.58

Tablica 4.4: Wyniki wyszukiwań algorytmem Fringe Search przy zastosowaniu różnych heurystyk

Dobór heurystyki dla FS ma największy wpływ. Umożliwa na około 25% przyspieszenie algorytmu, ale niewłaściwy wybór skazuje na nawet kilkukrotnie dłuższe obliczenia. Łamanie remisów nie sprawdza się i powoduje, że więcej wierzchołków jest rozpatrywanych w każdej iteracji, przez co nieuporządkowana otwarta lista, przeszukiwana jest więcej razy, co prowadzi do znaczącego wydłużenia czasu wyszukiwania. Najlepiej sprawdza się tutaj heurystyka Manhattan, z uwagi na najmniej kosztowne obliczenia, podczas sprawdzania czy aktualny wierzchołek ma zostać rozpatrzony, co wymaga wyliczenia oszacowania. Statystyczne średnie odchylenie czasu wyszukiwania przyjmuje duże wartości, a w przypadku heurystyki ukośnej jest ono nawet

większe od średniego czasu wyszukiwania. Świadczy to o znacznie wydłużonym czasie działania w niektórych przypadkach. Jest to zły znak, jeżeli algorytm ma zostać użyty w grze komputerowej. Może powodować bowiem nieoczekiwane spowolnienie rozgrywki.

4.2 Porównanie algorytmów dla zbioru map z gry Baldur's Gate II

algorytm	najlepsza heurystyka	długość znal. ścieżki	czas wyszukiw. w ms	st. odchyl. średniego czasu wyszukiw.	rozpatrzone wierzchołki	odwiedzone wierzchołki	największa ilość elementów OPEN listy
A*	Octile * 1.75	204.862	1.91146	1.08861	3067.57	5914.74	547.416
HPA*	Manhattan	233.215	0.389639	0.161782	548.889	1179.46	110.361
JPS	Octile * 1.75	203.025	0.801874	0.508974	34.7655	18545.1	9.5569
FS	Manhattan	210.771	7.76984	7.42448	8919.2	351974	287.281

Tablica 4.5: Porównanie algorytmów dla zbioru map z gry Baldur's Gate II

Liderem rankingu pod względem czasu wyszukiwania jest zdecydowanie HPA*. Jednak znaleziona przez niego ścieżka jest około 16% dłuższa od optymalnej. W celu jej poprawy można zastosować algorytmy wygładzające. Jeżeli wymogiem jest wysoka jakość ścieżki, o wiele lepszym rozwiązaniem jest użycie JPS. Czas działania jest około dwa razy dłuższy, ale nadal mieści się w granicy 1 ms. Dodatkowym plusem jest praktycznie zerowe, dodatkowe zapotrzebowanie na pamięć, o czym świadczy niska, średnia największa liczba elementów w otwartej liście. FS działa kilkukrotnie wolniej, niż A* na danym zbiorze map, jednak zużywa dwukrotnie mniej pamięci.

Z uwagi na bardzo wysoką wartość odchylenia standardowego która mogła świadczyć o lepszym działaniu FS, względem A* na wyszukiwaniach o krótszej ścieżce między punktami, przeprowadzone zostały dodatkowe testy. Użyty został zbiór 120 map, również z gry Baldur's Gate II, o rozdzielczości map między 50×50 a 320×320 płytek. W tabeli przedstawione zostały najkorzystniejsze wyniki spośród przeprowadzonych eksperymentów.

algorytm	najlepsza heurystyka	długość znal. ścieżki	czas wyszukiw. w ms	st. odchyl. średniego czasu wyszukiw.	rozpatrzone wierzchołki	odwiedzone wierzchołki	największa ilość elementów OPEN listy
A*	Octile	54.4881	0.215001	0.377774	677.01	1056.96	173.473
A*	Octile * 1.75	55.5978	0.162661	0.286932	374.314	704.506	139.422
FS	Manhattan	56.2088	0.772288	1.61414	891.587	26227	76.2101

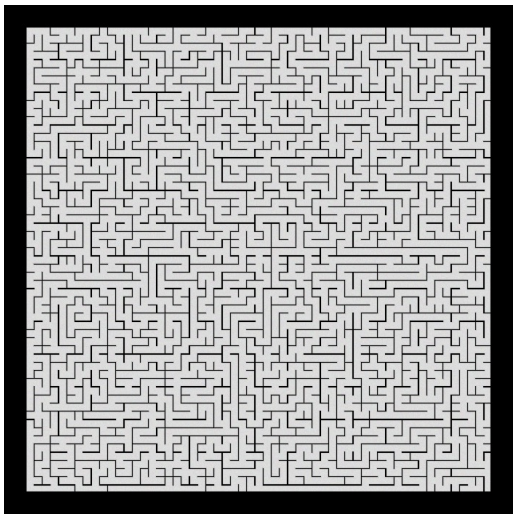
Tablica 4.6: Porównanie algorytmów dla zbioru map z gry Baldur's Gate II, o mniejszej rozdzielczości plansz

Na mapach o krótszej długości ścieżki również nie ma zauważalnej poprawy w czasie działania Fringe Search, względem A*.

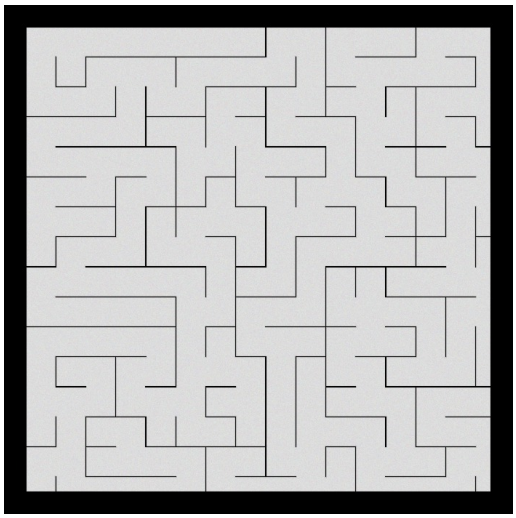
4.3 Porównanie algorytmów na zbiorze map labiryntów

Zbiór składa się z 60 map w postaci labiryntów, o zmiennej szerokości między sąsiednimi ścianami, od szerokości 1 płytki, aż po 32. Przykładowe mapy z tego zbioru pokazano na rysunkach 4.4 oraz 4.5.

Nowa wartość zawyżonego oszacowania została wyznaczona w analogiczny sposób, jak dla wcześniejszego zbioru map, jest identyczna jak poprzednia i wynosi 1.75.



Rysunek 4.4: Przykład labiryntu o szerokości 8



Rysunek 4.5: Przykład labiryntu o szerokości 32

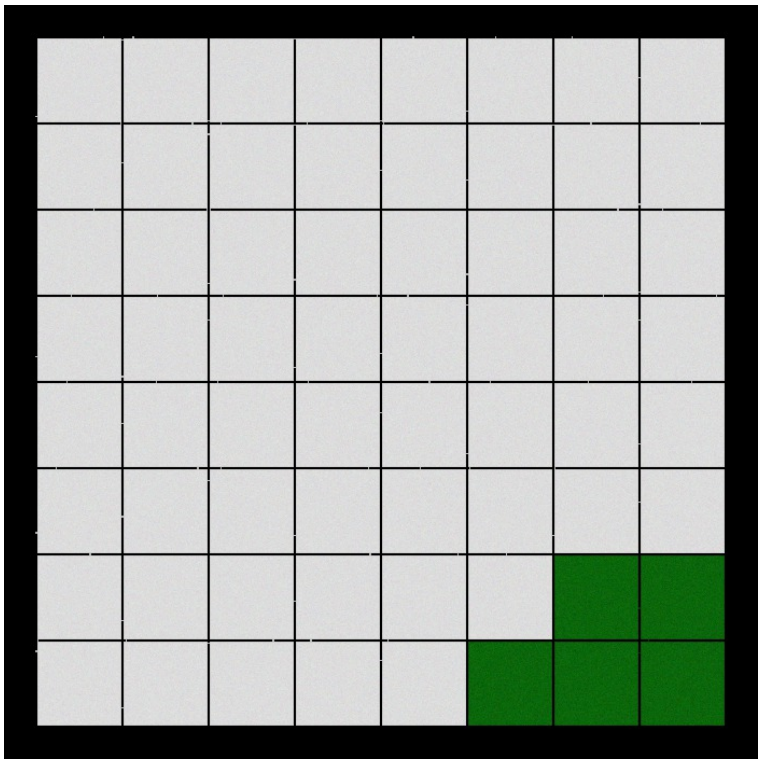
algorytm	najlepsza heurystyka	długość znal. ścieżki	czas wyszukiw. w ms	st. odchyl. średniego czasu wyszukiw.	rozpatrzone wierzchołki	odwiedzone wierzchołki	największa ilość elementów OPEN listy
A*	Octile	1453.32	13.1203	6.30837	54649.7	68358.8	644.828
A*	Octile * 1.75	1497.48	13.2415	6.40639	48847.6	70327.7	540.931
HPA*	Octile * 1.75	1701.2	2.87617	1.46995	6869.24	9553.93	88.8708
JPS	Octile * 1.75	1484.34	6.47928	6.37343	3751.38	47809.1	19.121
FS	Manhattan	1464.17	57.593	44.9948	64481.9	2740410	376.849

Tablica 4.7: Porównanie algorytmów na zbiorze map labiryntów

Na tym zbiorze liderem pod względem czasu wyszukiwania jest HPA*, jednak znaleziona ścieżka jest średnio 17% dłuższa względem optymalnej. Algorytm działa szybciej, gdyż wyznaczone rozwiązanie na wyższym poziomie abstrakcji nakierowuje algorytm w dobrym kierunku, więc nie musi on podążać na ślepo. Dobór heurystyki na tej mapie ma dużo mniejsze znaczenie, a może nawet osłabić algorytm. Widać to na przykładzie A*, gdzie agresywniejsze oszacowanie pogorszyło zarówno czas wyszukiwania, jak i długość znalezionej ścieżki. Nakierowywanie algorytmu w kierunku celu, na mapie z labiryntem, na niewiele się zdaje, gdyż prawdziwa ścieżka często podąża w zupełnie przypadkowych kierunkach.

4.4 Zbiór map pokoi

Zbior składa się z 40 map pokoi o rozdzielczości 512×512 , gdzie większość pokoi jest pusta. Wyznaczone obszary można pokonywać przez małe przejścia między nimi. Przykładową mapę z tego zbioru pokazano na rysunku 4.6.



Rysunek 4.6: Przykładowa mapa zbioru pokoi

algorytm	najlepsza heurystyka	długość znal. ścieżki	czas wyszukiw. w ms	st. odchyl. średniego czasu wyszukiw.	rozpatrzone wierzchołki	odwiedzone wierzchołki	największa ilość elementów OPEN listy
A*	Octile	234.542	4.44474	1.31744	11386.8	18031.3	1056.22
A*	Octile * 1.65	252.453	2.31356	1.29877	3781.33	8243.86	818.396
HPA*	Manhattan	276.267	0.751691	0.130777	1202.27	2227.65	204.764
JPS	Octile * 1.65	242.249	0.458842	0.339266	52.0825	10154.2	23.7124
FS	Manhattan	236.126	12.573	7.91916	12908.3	725569	452.268

Tablica 4.8: Porównanie algorytmów na zbiorze map pokoi

Na przykładzie tego zbioru widać jak duży wpływ ma wybór odpowiedniej heurystyki. Algorytm A* udało się przyspieszyć prawie dwukrotnie, a liczbę rozpatrywanych wierzchołków zmniejszyć ponad trzykrotnie. Na tym zbiorze map najlepiej radzi sobie JPS, wyprzedzając HPA* o około 60%. Dzieje się tak z uwagi na idealne warunki dla JPS, czyli istnienie sporych, pustych obszarów, z wieloma symetrycznymi ścieżkami, które mogą zostać łatwo odcięte. FS jest około 2.8 razy gorszy, niż A* i jest to jego najlepszy rezultat spośród rozpatrywanych zbiorów map.

4.5 Zużycie pamieci HPA*

Zbiór map	Wielkość grafu podstawowej mapy w kB	Wielkość pierwszego poziomu abstrakcji w HPA* w kB	Procentowy dodatkowy narzut pamięci
Baldur's Gate II 512x512	18200,11	546,84	3,0%
Labirynty 512x512	10847,78	1729,64	16,0%
Pokoje 512x512	26786,6	2390,7	8,9%

Tablica 4.9: Zużycie pamięci HPA* na różnych zbiorach map

Ilość dodatkowej pamięci potrzebnej do przetrzymywania abstrakcyjnego grafu dla HPA* jest zależna od ilości przeszkód na mapie, ponieważ od nich zależy liczba wejść do klastrów, które należy stworzyć. Zbiór map z gry Baldur's Gate II, w przeciwieństwie do zbioru z labiryntami zawiera wiele pustych przestrzeni, dla których generowana jest mniejsza liczba wejść. Mimo, że pokoje są obszarami bez przeszkód w ich wnętrzu, w zbiorze znajduje się spora ich liczba, o powierzchni mniejszej od klastra. Skutkuje to większym narzutem pamięci względem zbioru map z gry.

Projekt i implementacja algorytmów

5.1 Opis technologii

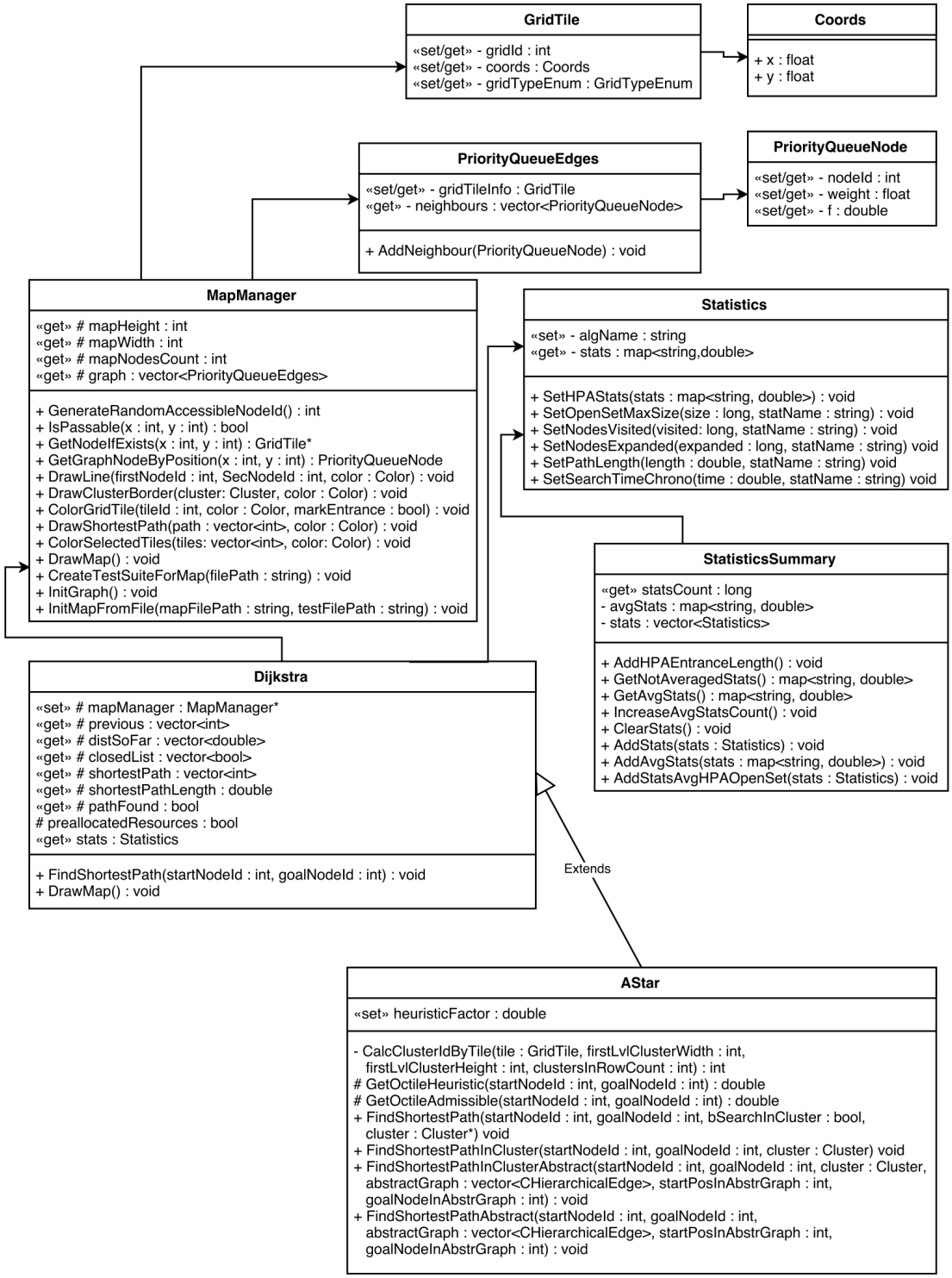
Projekt zaimplementowano przy użyciu języka C++. Wykorzystano bibliotekę standardową w wersji C++14 [2] [3] , bibliotekę boost w wersji 1.65.1 [1] oraz freeglut w wersji 3.0.0 [4]. Wykorzystany został kompilator MSVC w wersji 14.11.25503. Testy przeprowadzane były w wersji release, po włączeniu maksymalnych optymalizacji podczas kompilacji. [7]

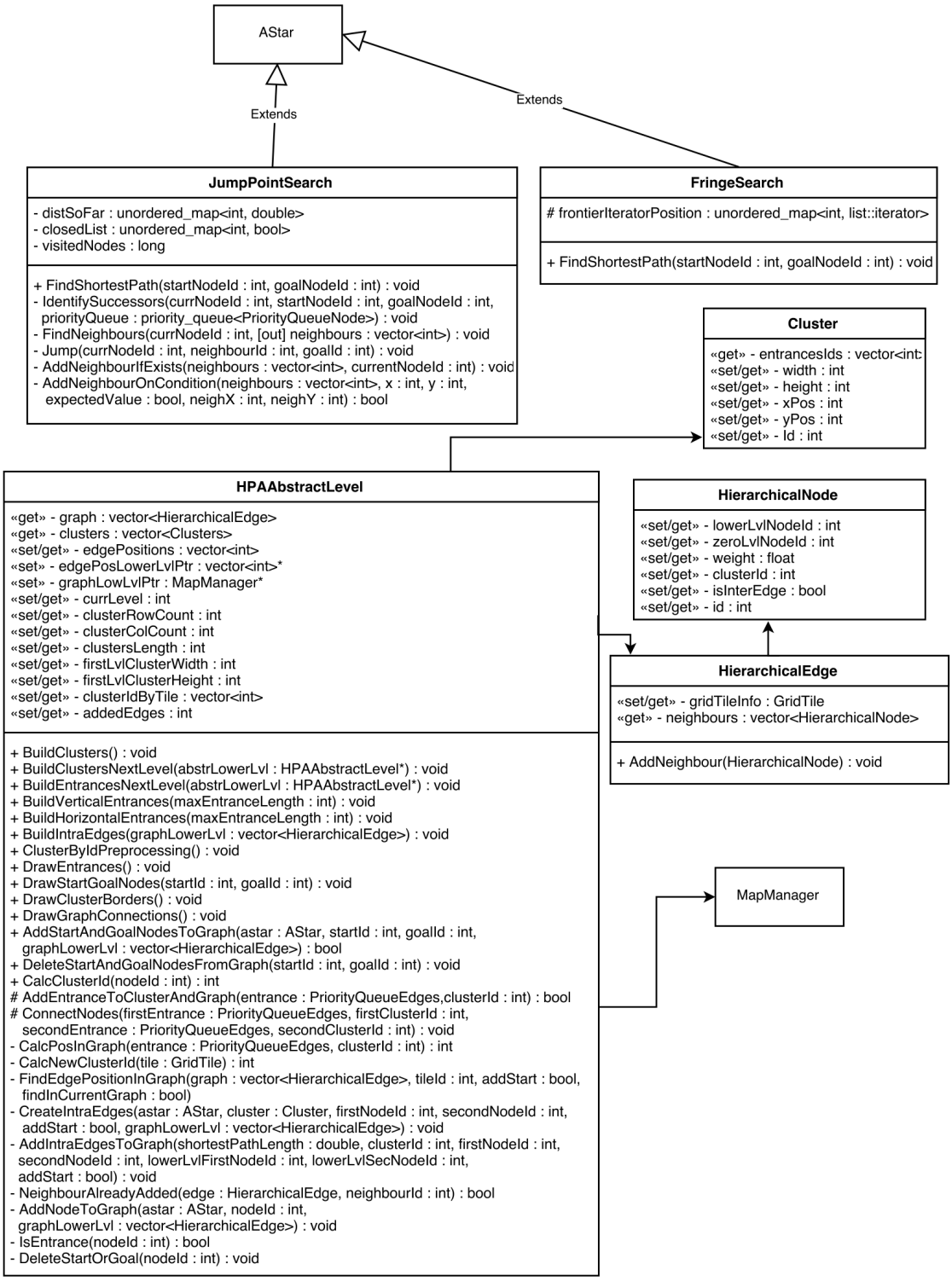
5.2 Opis klas

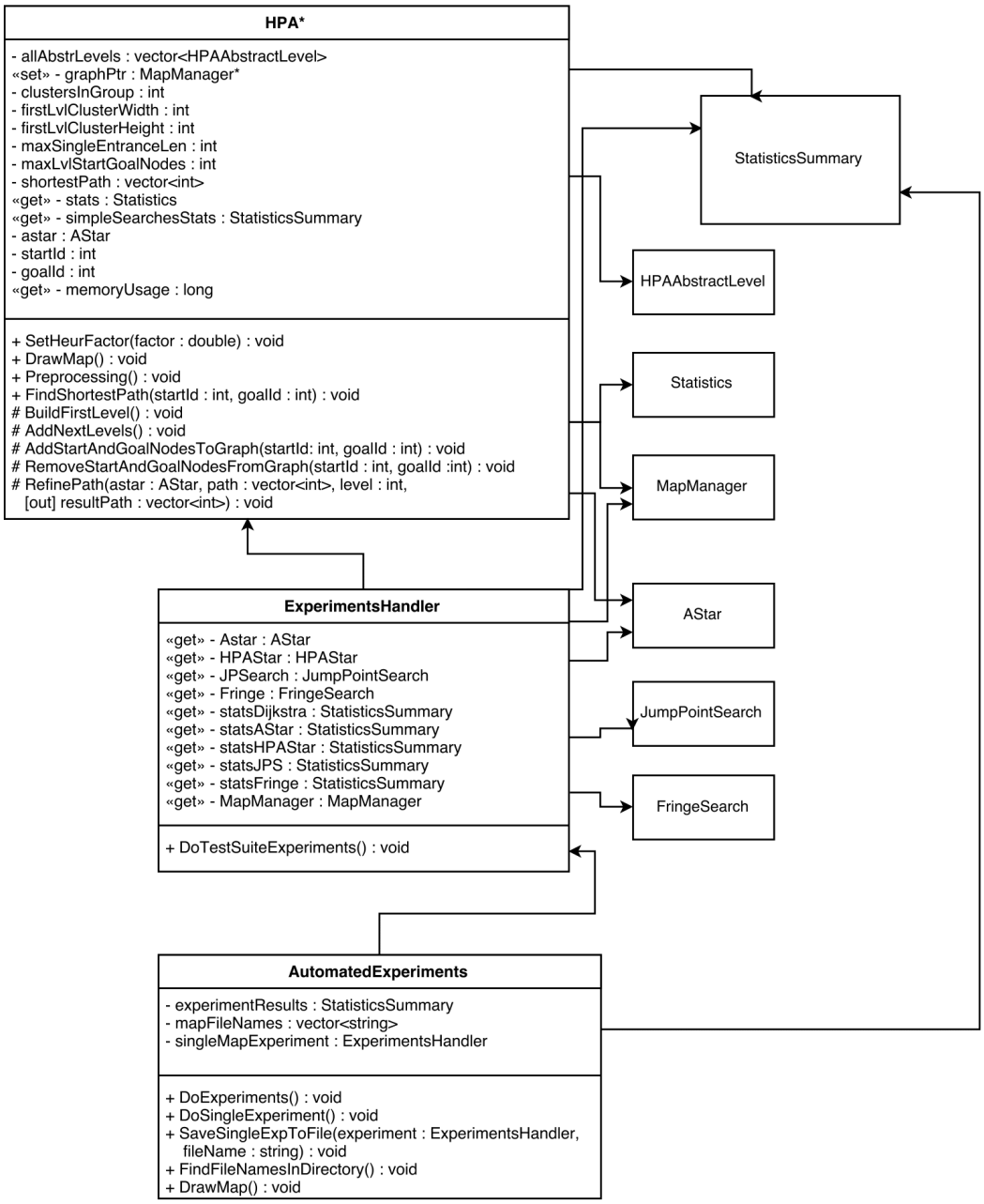
- **Coords** – przechowuje współrzędne płytek na mapie.
- **GridTile** – przechowuje współrzędne płytki i informację o jej rodzaju.
- **PriorityQueueNode** – klasa reprezentująca wierzchołek znajdujący się w kolejce priorytetowej, jak również wierzchołek w głównym grafie.
- **PriorityQueueEdges** – przechowuje wszystkie krawędzie wychodzące z wierzchołka.
- **MapManager** – klasa odpowiedzialna za wczytanie mapy z pliku, stworzenie jej reprezentacji w postaci grafu i przechowywanie go. Podczas wczytywania mapy, w razie potrzeby, generowane są dla niej punkty startowe i końcowe, które posłużą do testów algorytmów. Implementuje metody potrzebne do rysowania składowych mapy.
- **Statistics** – przechowuje i zbiera informacje o statystykach z pojedynczego wyszukiwania przeprowadzonego przez algorytm.
- **StatisticsSummary** – przechowuje i zbiera informacje o statystykach ze wszystkich wyszukiwań przeprowadzonych danym algorytmem.
- **Dijkstra** – klasa implementująca algorytm Dijkstry.
- **AStar** – dziedziczy po klasie Dijkstry, odpowiada za podstawowe wyszukiwanie ścieżki na mapie. Używana przez algorytm HPA* do wyznaczania ścieżek w klastrach oraz na kolejnych poziomach abstrakcji, zarówno w klastrze, jak i między nimi.
- **JumpPointSearch** – dziedziczy po AStar, implementuje algorytm Jump Point Search.
- **FringeSearch** – dziedziczy po AStar, implementuje algorytm Fringe Search.
- **Cluster** – przechowuje informacje o klastrach używanych przez algorytm HPA*, m.in. zawiera numery id wierzchołków będących punktami tranzytowymi.
- **HierarchicalNode** – odpowiednik PriorityQueueNode dla hierarchicznego wyszukiwania.
- **HierarchicalEdge** – odpowiednik PriorityQueueEdges dla hierarchicznego wyszukiwania.

- **HPAAbstractLevel** – klasa odpowiedzialna za tworzenie dodatkowych poziomów abstrakcji dla algorytmu HPA*, implementuje wszystkie składowe wykorzystywane podczas preprocessingu. Są nimi: budowa klastrów, wyznaczanie punktów tranzytowych, tworzenie krawędzi wewnątrz oraz międzyklastrowych. Odpowiada również za dodawanie i usuwanie punktów startowych i końcowych podczas właściwego etapu wyszukiwania przez HPA*. Implementuje metody odpowiedzialne za rysowanie powyższych elementów.
- **HPA** – implementuje algorytm HPA*. Do wyszukiwania na abstrakcyjnych poziomach i wewnątrz klastrów używa algorytmu A*.
- **ExperimentsHandler** – odpowiada za przeprowadzanie eksperymentów dla każdego z zaimplementowanych algorytmów na wskazanej mapie.
- **AutomatedExperiments** – przeprowadza eksperymenty dla każdej mapy znajdującej się w odpowiednim folderze, zebrane wyniki zapisywane są do pliku.

5.2.1 Diagramy klas




AStar
Extends
Extends
JumpPointSearch
- distSoFar : unordered_map<int, double>
- closedList : unordered_map<int, bool>
- visitedNodes : long
+ FindShortestPath(startNodeId : int, goalNodeId : int) : void
- IdentifySuccessors(currNodeId : int, startNodeId : int, goalNodeId : int, priorityQueue : priority_queue<PriorityQueueNode>) : void
- FindNeighbours(currNodeId : int, [out] neighbours : vector<int>) : void
- Jump(currNodeId : int, neighbourId : int, goalId : int) : void
- AddNeighbourIfExists(neighbours : vector<int>, currentNodeId : int) : void
- AddNeighbourOnCondition(neighbours : vector<int>, x : int, y : int, expectedValue : bool, neighX : int, neighY : int) : bool
FringeSearch
frontierIteratorPosition : unordered_map<int, list::iterator>
+ FindShortestPath(startNodeId : int, goalNodeId : int) : void
Cluster
«get» - entrancesIds : vector<int>
«set/get» - width : int
«set/get» - height : int
«set/get» - xPos : int
«set/get» - yPos : int
«set/get» - Id : int
HPAAbstractLevel
«get» - graph : vector<HierarchicalEdge>
«get» - clusters : vector<Clusters>
«set/get» - edgePositions : vector<int>
«set» - edgePosLowerLvlPtr : vector<int>*
«set» - graphLowLvlPtr : MapManager*
«set/get» - currLevel : int
«set/get» - clusterRowCount : int
«set/get» - clusterColCount : int
«set/get» - clustersLength : int
«set/get» - firstLvlClusterWidth : int
«set/get» - firstLvlClusterHeight : int
«set/get» - clusterIdByTile : vector<int>
«set/get» - addedEdges : int
+ BuildClusters() : void
+ BuildClustersNextLevel(abstrLowerLvl : HPAAbstractLevel*) : void
+ BuildEntrancesNextLevel(abstrLowerLvl : HPAAbstractLevel*) : void
+ BuildVerticalEntrances(maxEntranceLength : int) : void
+ BuildHorizontalEntrances(maxEntranceLength : int) : void
+ BuildIntraEdges(graphLowerLvl : vector<HierarchicalEdge>) : void
+ ClusterByIdPreprocessing() : void
+ DrawEntrances() : void
+ DrawStartGoalNodes(startId : int, goalId : int) : void
+ DrawClusterBorders() : void
+ DrawGraphConnections() : void
+ AddStartAndGoalNodesToGraph(astar : AStar, startId : int, goalId : int, graphLowerLvl : vector<HierarchicalEdge>) : bool
+ DeleteStartAndGoalNodesFromGraph(startId : int, goalId : int) : void
+ CalcClusterId(nodeId : int) : int
# AddEntranceToClusterAndGraph(entrance : PriorityQueueEdges,clusterId : int) : bool
# ConnectNodes(firstEntrance : PriorityQueueEdges, firstClusterId : int, secondEntrance : PriorityQueueEdges, secondClusterId : int) : void
- CalcPosInGraph(entrance : PriorityQueueEdges, clusterId : int) : int
- CalcNewClusterId(tile : GridTile) : int
- FindEdgePositionInGraph(graph : vector<HierarchicalEdge>, tileId : int, addStart : bool, findInCurrentGraph : bool)
- CreateIntraEdges(astar : AStar, cluster : Cluster, firstNodeId : int, secondNodeId : int, addStart : bool, graphLowerLvl : vector<HierarchicalEdge>) : void
- AddIntraEdgesToGraph(shortestPathLength : double, clusterId : int, firstNodeId : int, secondNodeId : int, lowerLvlFirstNodeId : int, lowerLvlSecNodeId : int, addStart : bool) : void
- NeighbourAlreadyAdded(edge : HierarchicalEdge, neighbourId : int) : bool
- AddNodeToGraph(astar : AStar, nodeId : int, graphLowerLvl : vector<HierarchicalEdge>) : void
- IsEntrance(nodeId : int) : bool
- DeleteStartOrGoal(nodeId : int) : void
HierarchicalNode
«set/get» - lowerLvlNodeId : int
«set/get» - zeroLvlNodeId : int
«set/get» - weight : float
«set/get» - clusterId : int
«set/get» - isInterEdge : bool
«set/get» - id : int
HierarchicalEdge
«set/get» - gridTileInfo : GridTile
«get» - neighbours : vector<HierarchicalNode>
+ AddNeighbour(HierarchicalNode) : void
MapManager


HPA*
- allAbstrLevels : vector<HPAAbstractLevel>
«set» - graphPtr : MapManager*
- clustersInGroup : int
- firstLvlClusterWidth : int
- firstLvlClusterHeight : int
- maxSingleEntranceLen : int
- maxLvlStartGoalNodes : int
- shortestPath : vector<int>
«get» - stats : Statistics
«get» - simpleSearchesStats : StatisticsSummary
- astar : AStar
- startId : int
- goalId : int
«get» - memoryUsage : long
+ SetHeurFactor(factor : double) : void
+ DrawMap() : void
+ Preprocessing() : void
+ FindShortestPath(startId : int, goalId : int) : void
# BuildFirstLevel() : void
# AddNextLevels() : void
# AddStartAndGoalNodesToGraph(startId: int, goalId : int) : void
# RemoveStartAndGoalNodesFromGraph(startId : int, goalId :int) : void
# RefinePath(astar : AStar, path : vector<int>, level : int,
[out] resultPath : vector<int>) : void
StatisticsSummary
HPAAbstractLevel
Statistics
MapManager
AStar
JumpPointSearch
FringeSearch
ExperimentsHandler
«get» - Astar : AStar
«get» - HPAStar : HPAStar
«get» - JPSearch : JumpPointSearch
«get» - Fringe : FringeSearch
«get» - statsDijkstra : StatisticsSummary
«get» - statsAStar : StatisticsSummary
«get» - statsHPAStar : StatisticsSummary
«get» - statsJPS : StatisticsSummary
«get» - statsFringe : StatisticsSummary
«get» - MapManager : MapManager
+ DoTestSuiteExperiments() : void
AutomatedExperiments
- experimentResults : StatisticsSummary
- mapFileNames : vector<string>
- singleMapExperiment : ExperimentsHandler
+ DoExperiments() : void
+ DoSingleExperiment() : void
+ SaveSingleExpToFile(experiment : ExperimentsHandler,
fileName : string) : void
+ FindFileNamesInDirectory() : void
+ DrawMap() : void

5.3 Omówienie kodów źródłowych

W tym podrozdziale, dla każdego z algorytmów, omówiona zostanie metoda odpowiedzialna za proces wyszukiwania ścieżki. Implementacje poszczególnych metod można znaleźć na dołączonej płycie CD w folderze `implementacja`.

Wyszukiwanie ścieżki przy pomocy algorytmu A*.

Kod źródłowy znajduje się w pliku `CAstar.cpp`. Sygnatura omawianej metody to:
```
void FindShortestPath(int iStartNode, int iGoalNode, bool bSimpleSearch,
CCluster *clusterPtr);
```

W liniach 46:76 następuje inicjalizacja zmiennych. W liniach 79:80 stworzenie punktu startowego i dodanie go do kolejki priorytetowej. Zmienna `bFoundBetterNeighbour` z wiersza 87 jest flagą, określającą, czy podczas rozpatrywania sąsiadów danego wierzchołka, znaleziony został następnik z f-wartością mniejszą lub równą, dotychczasowej najmniejszej w kolejce priorytetowej. Powyższy warunek sprawdzany jest w linii 147. Adres znalezionego w ten sposób wierzchołka zapamiętywany jest we wskaźniku w zmiennej `directNextNodePtr`. Wykonanie powyższych operacji umożliwia uniknięcie dodawania wierzchołka do kolejki priorytetowej, a w kolejnej iteracji, zdejmowanie go. W takim przypadku wykonywane są dwie kosztowne operacje mniej. Jeżeli taki wierzchołek zostanie znaleziony, informacje o nim odzyskiwane są w liniach 109:110, w przeciwnym przypadku następuje tradycyjne zdjęcie wierzchołka z kolejki, widoczne w linijkach 100:105. Jeżeli wierzchołek nie został już wcześniej rozpatrzony (linia 113), ani nie jest punktem docelowym (linia 122), rozpatrywani są wszyscy jego sąsiedzi (linie 129:130). Następnik rozpatrywany jest tylko, jeśli nie został rozpatrzony wcześniej(linia 134) i odległość do niego jest większa, niż nowo wyznaczony koszt podróży(linia 135).

Wyszukiwanie ścieżki przy pomocy algorytmu HPA*.

Kod źródłowy znajduje się w pliku `CHPAstar.cpp`. Sygnatura omawianej metody to:
```
void FindShortestPath(int iStartId, int iGoalId);
```

Na początku, do grafu dodawany jest punkt startowy i końcowy (linia 63). Zmienna `m_iMaxStartLvl` informuje, od którego poziomu abstrakcji należy rozpocząć wyszukiwanie ścieżki. Może bowiem zajść sytuacja, kiedy dla grafu z dodatkowym, jednym poziomem abstrakcji, wierzchołek startowy i końcowy dodane zostaną do tego samego klastra na najniższym poziomie. Wtedy wyszukiwanie należy przeprowadzić, tylko na najniższym poziomie. Zmienna `edgePos` informuje o położeniu wierzchołka o danym Id z grafu na najniższym poziomie, w grafie na aktualnie rozpatrywanym poziomie. W linii 82 znajdywana jest ścieżka na odpowiednio najwyższym poziomie abstrakcji. Kolejność wierzchołków na znalezionej ścieżce jest odwracana(linia 93), aby znajdywały się one w tablicy, w kolejności od startu do celu. Następnie znaleziona ścieżka jest uszczegóławiana(linia 97) na kolejno niższe poziomy abstrakcji(pętla w linii 95). Na koniec, z grafu usuwany jest punkt startowy i końcowy(linia 103).

Uszczegóławianie abstrakcyjnej ścieżki w algorytmie HPA*.

Kod źródłowy znajduje się w pliku `CHPAstar.cpp`. Sygnatura omawianej metody to:
```
void RefinePath(CAstar &AStar, std::vector<int> &path, int iLvl,
std::vector<int> &resultPath)
```

W pętli(linia 117) sprawdzane są kolejne pary wierzchołków, między którymi należy wyznaczyć ścieżkę na niższym poziomie abstrakcji. Ścieżka wyznaczana jest tylko w przypadku, kiedy oba wierzchołki należą do tego samego klastra(warunek w linii 119). W przeciwnym wypadku, wierzchołki są punktami tranzytowymi, między którymi nie istnieje ścieżka na niższym poziomie abstrakcji. Uszczegóławianie odbywa się odpowiednio na najniższym poziomie(linia 126) i przekazywany jest wskaźnik do klastra `clusterPtr`, w którym znajdują się oba wierzchołki(linia 125), albo na wyższym poziomie (linia 135). Znaleziona ścieżka, dodawana jest do wynikowej tablicy `resultPath`, w kolejności od punktu startowego do celu. Warunek z linii 145 zapewnia, że

do `resultPath` nie dodane zostaną, ani punkt startowy, ani końcowy, które dodawane są kolejno w liniach 115 i 151.

Wyszukiwanie ścieżki przy pomocy algorytmu Jump Point Search.

Wyszukiwanie algorytmem Jump Point Search różni się od A* wyborem wierzchołków, które należy dodać do kolejki priorytetowej w celu dalszego rozpatrzenia. Kod źródłowy znajduje się w pliku `CJPSearch.cpp`. Sygnatura omawianej metody to:

```
void IdentifySuccessors(int iCurrNode, int iStartNode, int iGoalNode,
std::priority_queue<CPriorityQueueNode, std::vector<CPriorityQueueNode>,
ComparePriorityQueueNode> &dijQueue)
```

Funkcja `FindNeighbours` z linii 116, przy zastosowaniu zasad odcinania sąsiadów, określa, którzy z sąsiadów wierzchołka o identyfikatorze `iCurrNode` mają zostać rozpatrzeni. Dla każdego z nich wyznaczane są punkty skoku, przez rekurencyjną funkcję `Jump` z linii 120. Na podstawie kierunku podróży między wierzchołkami `iCurrNode` i `iNeighbour` oraz przy zastosowaniu zasad odcinania sąsiadów, znajduje ona punkt skoku. W przypadku braku jego istnienia, bądź jeśli został on już wcześniej odwiedzony (warunki z linii 132) rozpatrywanie przechodzi dla kolejnego wyznaczonego sąsiada z `neighbours`. Kolejne kroki są analogiczne, jak dla algorytmu A*. Jedyną różnicą jest sposób wyznaczenia wagi krawędzi, między aktualnie rozpatrywanym wierzchołkiem `iCurrNode`, a punktem skoku `iJumpNode`. Wyznaczana jest ona, przy pomocy dopuszczalnej heurystyki ukośnej w linii 125. Funkcja gwarantuje podanie poprawnej odległości między wierzchołkami z uwagi na brak przeszkód między nimi, co z kolei gwarantuje funkcja `Jump`.

Wyszukiwanie ścieżki przy pomocy algorytmu Fringe Search.

Kod źródłowy znajduje się w pliku `CFringeSearch.cpp`. Sygnatura omawianej metody to:
`void FindShortestPath(int iStartNode, int iGoalNode)`

Z uwagi na brak uporządkowania elementów w liście dwukierunkowej `frontierList`, w każdej iteracji głównej pętli (linia 63), rozpatrywany jest każdy wierzchołek na tej liście. Jeżeli f-wartość `newFValue` danego wierzchołka jest większa od granicy `fLimit` w aktualnej iteracji głównej petli, rozpatrywany zostaje kolejny wierzchołek. W przeciwnym przypadku, jeśli wierzchołek nie jest celem (warunek w linii 82), analogicznie jak w A*, rozpatrywani są wszyscy jego sąsiedzi `neighbours`. Następnik `neigh`, dodawany jest zaraz za aktualnie rozpatrywanym wierzchołkiem `frontierIter` z listy dwukierunkowej. W celu uniknięcia wielu kopii tego samego wierzchołka, z różnymi f-wartościami, należy określić, czy znajduje się on już na liście `frontierIter` i jeśli tak, to go z niej usunąć. W tym celu potrzebna jest informacja o jego położeniu na liście. Jest to możliwe dzięki przetrzymywaniu iteratora do każdego z wierzchołków na liście. Iterator dodawany jest w linii 107. Jeśli wierzchołek jest usuwany z listy (linia 101), usuwany jest również iterator do niego (linia 102).

Podsumowanie

W pracy zaimplementowano cztery algorytmy rozwiązujące problem wyszukiwania najkrótszej ścieżki. Każdy z algorytmów został przeanalizowany teoretycznie oraz doświadczalnie. Liderem pod względem czasu działania jest HPA*, będący najszybszym spośród zaimplementowanych algorytmów na dwóch zbiorach map. Na zbiorze z pokojami został wyprzedzony przez JPS. W przyszłości należałoby zmierzyć zachowanie obu algorytmów w bardziej realistycznej sytuacji, z pokojami zawierającymi pewne przeszkody. Najbardziej oszczędnym pamięciowo algorytmem okazał się JPS, którego zapotrzebowanie na dodatkową pamięć, było kilkadziesiąt razy mniejsze niż u A*. W celu poprawienia optymalności ścieżki, wygenerowanej przez HPA*, należałoby zaimplementować algorytm wygładzający. Wybrany algorytm nie powinien być zbyt skomplikowany obliczeniowo, aby narzut związany z poprawą jakości ścieżki, nie pogorszył znacznie czasu działania HPA*. FS okazał się nieporadnym algorytmem na każdym ze zbiorów testowych. Swoim czasem działania nie zbliżył się do gorszej wersji A* z podstawową, nie przeszacowującą heurystyką. Standardowe odchylenie czasu działania przyjmuje dużą wartość, czasem przekraczającą średni czas na danym zbiorze map. Korzyści wynikające z podwójnie mniejszego zapotrzebowania na pamięć względem lepszej wersji A*, nie są wystarczające, aby algorytm nadawał się do użycia w grze. Należałoby sprawdzić, jak FS zachowuje się, mając częściowo posortowaną otwartą listę. Można by skorzystać z hashmapy przechowującej przedziały f-wartości. Wtedy wyszukiwanie następnego elementu do rozpatrzenia, sprowadzałoby się do zajrzenia do określonego przedziału elementów. W poszczególnych przedziałach znajdowałaby się mniejsza ilość elementów względem całej otwartej listy, więc można przypuszczać, że wyszukiwanie byłoby szybsze. Dalszym krokiem mogłoby być uporządkowanie poszczególnych przedziałów za pomocą kolejek priorytetowych. Inną ciekawą drogą byłoby sprawdzenie zachowania się HPA*, używając zamiast A*, do lokalnego wyszukiwania w klastrach, algorytmu JPS. Mogłoby to jeszcze bardziej przyspieszyć działanie HPA*. Kolejną możliwą drogą rozwoju mogłaby być zmiana wewnętrznej reprezentacji mapy rozgrywki z kraty ośmiokątnej, na punkty nawigacyjne, generowane automatycznie, dla każdej mapy. Należałoby skonstruować algorytm, który sprytnie by je łączył, w celu minimalizacji czynnika rozgałęzienia.

Bibliografia

[1] Boost library. Web pages: http://www.boost.org/users/history/version˙1˙65˙1.html.

[2] C++14 news. Web pages: https://isocpp.org/wiki/faq/cpp14#cpp14-what.

[3] C++14 standard, news. Web pages: http://www.drdobbs.com/cpp/the-c14-standard-what-you-need-to-know/240169034.

[4] Freeglut library. Web pages: http://freeglut.sourceforge.net/.

[5] Heuristics in games. Web pages: http://theory.stanford.edu/˜amitp/GameProgramming/Heuristics.html.

[6] Map reprezentations. Web pages: http://theory.stanford.edu/˜amitp/GameProgramming/MapRepresentations.html.

[7] Msvc optimizations. Web pages: https://msdn.microsoft.com/en-us/library/59a3b321.aspx.

[8] Total war:rome 2 pathfinding approximations. Web pages: https://aigamedev.com/open/upcoming/spotlight-challenges-2014/.

[9] Visibility graphs generator. Web pages: https://www.redblobgames.com/pathfinding/visibility-graphs/.

[10] B. Anguelov. Video game pathfinding and improvements to discrete search on grid - based maps. 2011.

[11] A. Botea, M. Müller, J. Schaeffer. Near optimal hierarchical path-finding. 2006.

[12] I. Millington, J. Funge. *Artificial Intelligence For Games, Second Edition*. Morgan Kaufmann, 2009.

[13] M. Nosrati, R. Karimi, H. A. Hasanvand. Investigation of the * (star) search algorithms: Characteristics, methods and approaches. 2012.

[14] S. Russel, P. Norvig. *Artificial Intelligence: A Modern Approach, Third Edition*. Prentice Hall, 2010.

[15] N. R. Sturtevan. Benchmarks for grid-based pathfinding. 2012.

Zawartość płyty CD

W folderze `praca`, znajduje się praca inżynierska w pliku pdf. W folderze `implementacja` znajdują się kody źródłowe projektu.