Analiza i implementacja algorytmów odnajdywania ścieżki do zastosowania w grach przeglądarkowych

Analysis and implementation of pathfinding algorithms for use in browser games

Beata Laszkiewicz[1], Tomasz Sobczak[2]

STRESZCZENIE: Celem tego artykułu jest przedstawienie, porównanie oraz implementacja algorytmów odnajdywania ścieżki do zastosowania w grach przeglądarkowych z wykorzystaniem ogólnodostępnych, darmowych technologii internetowych. Pokazano również możliwość wykorzystania najlepszego algorytmu w grze przeglądarkowej.

ABSTRACT: The goal of this article is to present, compare and implement path finding algorithms for use in browser games, using public, free internet technologies. The possibility of using the best algorithm in a browser game is also shown.

SŁOWA KLUCZOWE: gra przeglądarkowa, graf, algorytm Bellmana-Forda, algorytm Dijkstry, algorytm A-star

KEY WORDS: browser game, graph, Bellman-Ford algorithm, Dijkstra algorithm, A-star algorithm

1. Wprowadzenie

Ciągły rozwój technologii sprawił, że niegdyś proste i statyczne strony internetowe, które nie pozwalały na żadną interakcję z użytkownikiem, dziś mogą reagować nawet na najmniejszy gest. W dobie Internetu, kiedy ogromną popularnością cieszą się portale i serwisy społecznościowe, swoje miejsce znalazł również przemysł gier. Dzięki takim projektom jak Facebook czy Google+ producenci uzyskali dynamicznie rosnący rynek zbytu – ogromną rzeszę osób, która w innym przypadku nie zainteresowałaby się ich produktami. Jednak gdy te stały się elementem ulubionych serwisów społecznościowych, zaczęły być odbierane jako coś wartościowego, w co warto spróbować zagrać.

Gry przeglądarkowe mogą być bardziej lub mniej zaawansowane, jednakże niezależnie od tego, w jakiej technologii zostały stworzone, bardzo często wykorzystują podobne algorytmy i wzorce projektowe. Jednym z typów takich właśnie algorytmów są algorytmy wyszukiwania ścieżki. Algorytmy odnajdywania ścieżek są ściśle powiązane z zagadnieniami sztucznej inteligencji oraz teorii grafów. Pierwotnie były wykorzystywane do odnajdywania najkrótszych odległości pomiędzy wierzchołkami grafu, ale wraz z rozwojem technologii i sztucznej inteligencji zaczęto ich używać m.in. przy wyszukiwaniu i planowaniu tras.

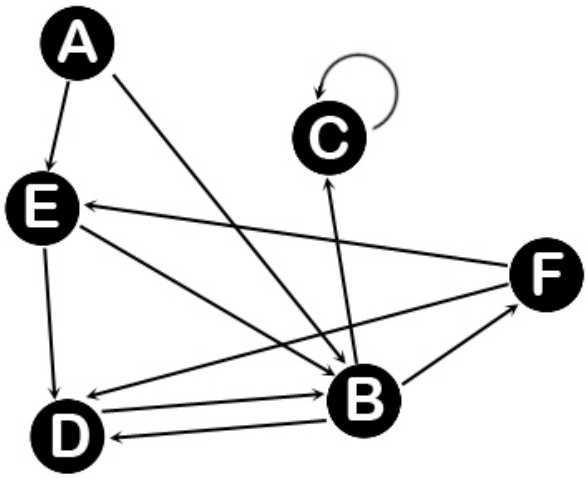
2. Podstawy teoretyczne

Teoria grafów to obszerny dział matematyki i informatyki, zajmujący się badaniem właściwości grafów. Grafy możemy podzielić na proste (grafy nieskierowane, grafy), skierowane (digrafy) oraz mieszane.

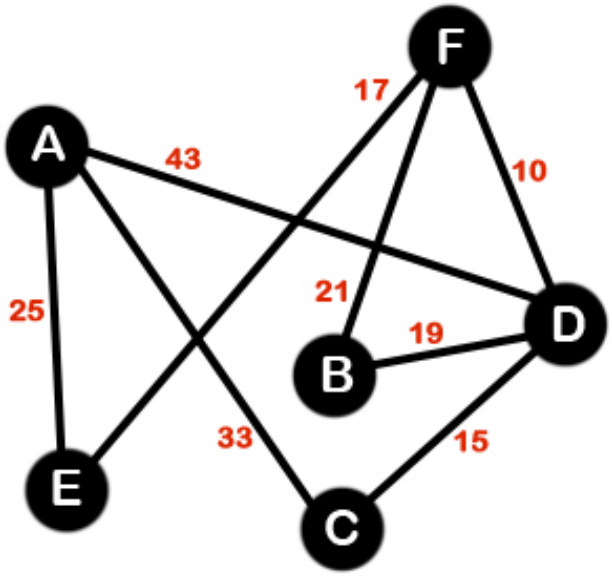
Graf to zbiór wierzchołków (węzłów), które mogą być połączone krawędziami w taki sposób, że każda krawędź kończy się i zaczyna w którymś z wierzchołków. Dodatkowo krawędzie mogą posiadać swoją wagę (koszt przejścia), którą najczęściej można wyobrazić sobie jako odległość lub czas przejścia pomiędzy wierzchołkami, jednakże ich interpretacje są nieograniczone. Dla każdego węzła możemy również określić jego stopień, czyli liczbę krawędzi incydentnych z tym węzłem. Graf skierowany charakteryzuje się krawędziami, dla których określony jest jeden kierunek. Takie krawędzie pozwalają przejść z jednego wierzchołka grafu do drugiego, ale nie na odwrót. Grafy mieszane jednocześnie posiadają zarówno krawędzie nieskierowane, jak i skierowane[3].

Na rys. 1 oraz rys. 2 przedstawiono przykłady grafów: skierowanego i nieskierowanego.

1. Wrocławska Wyższa Szkoła Informatyki Stosowanej we Wrocławiu, e-mail: blaszkiewicz@horyzont.eu, ORCID 0000-0002-1447-4755.
2. Wrocławska Wyższa Szkoła Informatyki Stosowanej we Wrocławiu, e-mail: tomasz-chudzik@wp.pl.
3. K. A. Ross, C. R. B. Wright, *Matematyka dyskretna*, Wydawnictwo Naukowe PWN, Warszawa 1996, s. 327–336. M. M. Sysło, N. Deo, J. S. Kowalik, *Algorytmy optymalizacji dyskretnej,* Wydawnictwo Naukowe PWN, Warszawa 1995, s. 179.

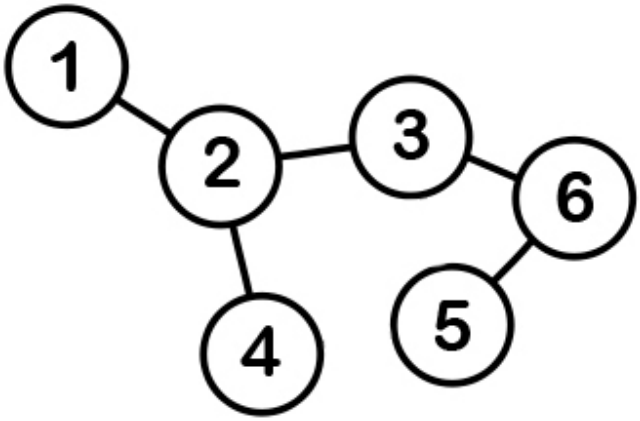


Rys. 1. Przykład grafu skierowanego[4]

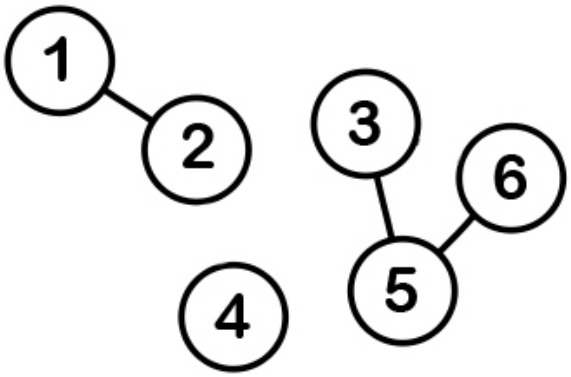


Rys. 2. Przykład grafu nieskierowanego wraz z wagami

Grafy można również podzielić ze względu na różne właściwości – tzw. klasy. Najczęściej spotykane z nich to grafy spójne, grafy acykliczne oraz drzewa. Z grafem spójnym mamy do czynienia, gdy między jego dowolnymi dwoma węzłami istnieje łącząca je ścieżka, czyli z dowolnego wierzchołka grafu można dotrzeć do każdego innego. Przykłady grafu spójnego i niespójnego znajdują się na rys. 3. i rys. 4.



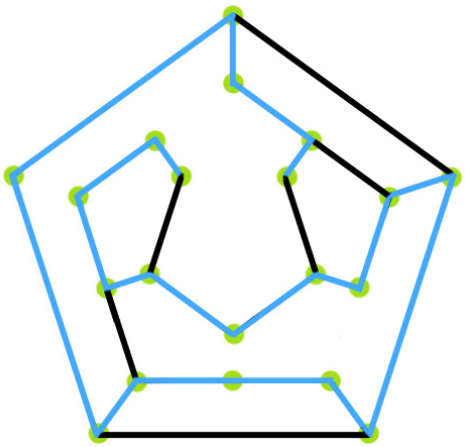
Rys. 3. Przykład grafu spójnego



Rys. 4. Przykład grafu niespójnego

Grafy acykliczne nie posiadają cykli, czyli ścieżek, których koniec pokrywa się początkiem, i nie zawierają innych powtarzających się węzłów lub krawędzi. Na rysunku 5. został przedstawiony graf z oznaczonym cyklem Hamiltona, czyli cyklem, którego ścieżka nie przechodzi przez żaden z wierzchołków dwukrotnie. Na rys. 6. zaprezentowano jego podgraf (graf powstały przez usunięcie z grafu nadrzędnego pewnej liczby wierzchołków lub krawędzi) tworzący graf acykliczny. Na niebiesko zaznaczono ścieżkę przechodzącą przez wszystkie wierzchołki, lecz nietworzącą cyklu (ścieżka nie została zamknięta).

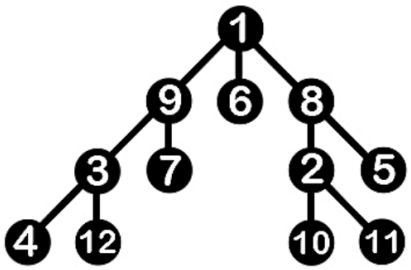
Rys. 5. Przykład grafu cyklicznego



Rys. 6. Przykład grafu acyklicznego

4. Wszystkie rysunki są autorstwa T. Sobczaka.

Drzewami nazywamy grafy, które są jednocześnie acykliczne oraz spójne. Drzewo, w którym jeden z wierzchołków został wyróżniony, nazywane jest drzewem ukorzenionym, a wyróżniony wierzchołek – korzeniem. Na rys. 7 został przedstawiony przykład drzewa ukorzenionego, którego korzeniem jest wierzchołek oznaczony cyfrą 1.



Rys. 7. Przykład drzewa ukorzenionego

2.1. Algorytm Bellmana-Forda

Algorytm ten wykorzystywany jest najczęściej do odnajdywania optymalnej trasy w sytuacjach, gdy nie liczy się czas, a wynik. Pierwotnie służył do odnajdywania ścieżek o najmniejszej wadze pomiędzy wszystkimi wierzchołkami danego grafu.
Algorytm opiera się na metodzie relaksacji, czyli sprawdzeniu, czy przy przejściu daną krawędzią grafu nie otrzymamy trasy krótszej (mniej kosztownej) niż dotychczas znana ścieżka. Niestety algorytm ze względu na swoją budowę posiada dużą złożoność czasową – O(|V|•|E|), gdzie V to zbiór wierzchołków, a E to zbiór krawędzi między wierzchołkami. Zaletą tego algorytmu jest możliwość użycia go dla grafu zawierającego ujemne wagi przejść między wierzchołkami[5]. Pełny opis algorytmu można znaleźć w pracy *Analiza i implementacja algorytmów odnajdywania ścieżki do zastosowania w grach przeglądarkowych*[6].

2.2. Algorytm Dijkstry

W przeciwieństwie do algorytmu Bellmana-Forda ten algorytm nie zezwala na ujemne koszty krawędzi między węzłami. Algorytm Dijkstry zachowuje się podobnie jak algorytmy zachłanne, ponieważ wybiera zawsze pole o najmniejszym koszcie (i nie powraca już do niego). Dzięki tym założeniom złożoność czasowa algorytmu jest dużo mniejsza: przy zastosowaniu do przechowywania informacji kopca binarnego, czyli struktury danych reprezentującej drzewo, w którym stopień każdego wierzchołka nie może być większy od 3, wynosi O((|E| log|V|), gdzie V to zbiór wierzchołków, a E to zbiór wszystkich krawędzi pomiędzy wierzchołkami[7]. Pseudokod oraz omówienie algorytmu można znaleźć w pracy *Analiza i implementacja [...]*[8].

2.3. Algorytm A*

Algorytm A* (A-Star) jest rozwinięciem algorytmu Dijkstry. Dzięki zastosowaniu heurystyki (szacowania) skrypt jest w stanie wyznaczyć optymalną trasę stosunkowo niewielkim kosztem czasowym. Przy zastosowaniu odpowiedniej heurystyki algorytm może osiągnąć wyniki w czasie istotnie mniejszym niż algorytm Dijkstry.
Algorytm przeszukuje graf (mapę) poprzez wędrówkę po ścieżce, która według obecnie znanych kosztów jest najniższa, jednocześnie zapamiętując alternatywne rozwiązania. Jeżeli jakiś element trasy osiągnie koszt wyższy niż alternatywne rozwiązanie, wtedy obecny kierunek jest porzucany na koszt tańszego rozwiązania. Proces ten trwa, dopóki cel nie zostanie osiągnięty.
W tabeli 1. przedstawiono dwie najczęściej wykorzystywane funkcje heurystyczne, które oszacowują odległość pomiędzy dwoma punktami. Zmienna A to punkt początkowy, zmienna B to punkt końcowy, a indeksy przy zmiennych oznaczają współrzędne punktu, czyli oznaczenie Ax to współrzędna x dla punktu A.

Nazwa funkcji	Wzór
Manhattan	$\mathbf{H} = \lvert B_x - A_x \rvert + \lvert B_y - A_y \rvert$
Odległość Euklidesa	$\mathbf{H} = \sqrt{(B_x - A_x)^2 + (B_y - A_y)^2}$

Tab. 1. Popularne funkcje heurystyczne

Pełny pseudokod algorytmu oraz jego szczegółowy opis można znaleźć w pracy Analiza i implementacja […][9].

3. Środowisko programistyczne

Dostępnych jest wiele języków programowania, które mogą zostać wykorzystane do implementacji przedstawionych wcześniej algorytmów, ale pod uwagę wzięto jedynie te darmowe i cieszące się w ostatnich czasach popularnością. Jeśli chodzi o sposób przechowywania danych, zastosowano rozwiązania korzystające z języka SQL (m.in. MySQL) oraz takie, które kierują się obecnie modną zasadą NoSQL (m.in. MongoDB).
Jako język programowania, w którym zaimplementowane zostaną przedstawione wcześniej algorytmy, wybrano JavaScript ze względu na jego uniwersalność. Jego zaletami są obsługa przez wszystkie obecne przeglądarki internetowe oraz możliwość użycia jednego języka programowania do stworzenia frontendu i backendu aplikacji.
Do przechowywania danych na serwerze wykorzystany

5. *Algorytmy i struktury danych*, wykład 7: http://hector.tu.kielce.pl/przedmioty/aisd-wyk/AiSD_w07.pdf, [dostęp: 30.11.2021].
6. T. Sobczak, *Analiza i implementacja algorytmów odnajdywania ścieżki do zastosowania w grach przeglądarkowych*, inżynierska praca dyplomowa, Wrocławska Wyższa Szkoła Informatyki Stosowanej, Wrocław 2012.
7. K. A. Ross, C. R. B. Wright, *op. cit.*, M. M. Sysło, N. Deo, J. S. Kowalik, *op. cit. Algorytmy i struktury danych*, *op. cit.*
8. T. Sobczak, *op. cit.*
9. *Ibidem.*

zostanie format JSON, gdyż dane nie muszą być przechowywane w sposób relacyjny i nie będą potrzebne kwerendy czy innego typu zapytania. Ponadto cechuje go prostota przekazania informacji do skryptu JavaScript – dzięki zastosowaniu takiego rozwiązania wszelkie informacje (np. o terenie) zostaną dostarczone do użytkownika aplikacji bardzo szybko.

4. Implementacja algorytmów

4.1. Algorytm Bellmana-Forda

W listingu 1. znajduje się główna pętla algorytmu, która jest odpowiedzialna za relaksację krawędzi. Przed rozpoczęciem pętli miało miejsce przygotowanie zmiennych. W przypadku, gdy nie ma już żadnych krawędzi, które mogłyby zostać zrelaksowane, pętla zostaje przerwana.
Zmienne wykorzystane w funkcji:
Vertex – zbiór wszystkich pól na mapie
Edges – zbiór krawędzi pomiędzy tymi polami

```
1. for (i in Vertex) {
 2.   r = false;
 3.   for (j in Edges) {
 4.     e = Edges[j];
 5.     if (Vertex[e.v].cost > Vertex[e.u].cost + e.w) {
 6.       Vertex[e.v].cost   = Vertex[e.u].cost + e.w;
 7.       Vertex[e.v].parent = e.u;
 8.       r = true;
 9.     }
10.   }
11.   if (!r) {
12.     break;
13.   }
14. }
```

Listing 1. Implementacja algorytmu Bellmana-Forda[10]

4.2. Algorytm Dijkstry

Algorytm Dijkstry wykorzystuje kopce binarne do reprezentacji zbiorów, co wpływa na zwiększenie wydajności. Algorytm działa do momentu, gdy aktualny węzeł trafi do zbioru S lub zbiór Q będzie pusty – to kolejny krok w kierunku optymalizacji.
Zmienne wykorzystane w skrypcie (zob. listing 2.):

- `Q` – zbiór odwiedzonych węzłów
- `S` – zbiór węzłów, które algorytm uznał za całkowicie sprawdzone
- `P` – tablica relacji zachodzących pomiędzy węzłami (połączenia typu rodzic-dziecko)
- `M` – tablica informacji na temat pola

```
 1. while (Q.size() > 0) {
 2.   m = Q.pop();
 3.   x = M[m.id].x;
 4.   y = M[m.id].y;
 5.   S[m.id] = true;
 6.   if (x == end_x && y == end_y) {
 7.     break;
 8.   }
 9.   for (i = 1; i <= 6; ++i) {
10.     switch (i) {
11.       case 1:
12.         k = x;
13.         l = y – 1;
14.         break;
15.       case 2:
16.         k = x + 1;
17.         l = y;
18.         break;
19.       case 3:
20.         k = x + 1;
21.         l = y + 1;
22.         break;
23.       case 4:
24.         k = x;
25.         l = y + 1;
26.         break;
27.       case 5:
28.         k = x – 1;
29.         l = y;
30.         break;
31.       case 6:
32.         k = x – 1;
33.         l = y – 1;
34.         break;
35.     }
36.     if (accessibility(k,l)) {
37.       n = map[k][l];
38.       if (!S[n.id]) {
39.         z = Q.info(n.id);
40.         if ( z ) {
41.           if (m.cost + n.cost < z.cost) {
42.             Q.remove(n.id);
43.             Q.push({ id : n.id, cost : m.cost + n.cost});
44.             P[n.id] = m.id;
45.           }
46.         } else {
47.           Q.push({ id : n.id, cost : m.cost + n.cost});
48.           P[n.id] = m.id;
49.         }
50.       }
51.     }
52.   }
53. }
```

Listing 2. Implementacja algorytmu Dijkstry

4.3. Algorytm A*

Algorytm A*, jak wspomniano wcześniej, jest bardzo podobny do algorytmu Dijkstry, dlatego też w listingu 3.

10. Wszystkie listingi są autorstwa T. Sobczaka.

przedstawiono tylko fragment zaczynający się od linii 36 – cały poprzedzający ją kod jest identyczny jak w algorytmie Dijkstry.
Wykorzystane zmienne również są podobne:

- `Q` – zbiór węzłów otwartych (odwiedzonych)
- `S` – lista węzłów zamkniętych (uznanych za sprawdzone)
- `P` – tablica relacji zachodzących między wierzchołkami

```
36.    if (accessibility(k,l)) {
37.      n = map[k][l];
38.      if (!S[n.id]) {
39.        z = Q.info(n.id);
40.        if ( z ) {
41.          g = m.g + M[n.id].cost;
42.          if (g < z.g) {
43.            h = heuristic(k, l, end_x, end_y);
44.            Q.remove(n.id);
45.            Q.push({ id : n.id, f : g + h, g : g, h: h });
46.            P[n.id] = m.id;
47.          }
48.        } else {
49.          h = heuristic(k, l, end_x, end_y);
50.          g = m.g + M[n.id].cost;
51.          Q.push({ id : n.id, f : g + h, g : g, h: h });
52.          P[n.id] = m.id;
53.        }
54.      }
55.    }
56.  }
57. }
```

Listing 3. Implementacja algorytmu A*

W listingu 4. przedstawiono funkcję heurystyczną bazującą na metodzie Manhattan oraz niedoszacowanym modelu heurystyki. Dzięki niedoszacowaniu funkcja zawsze zwróci najniższy możliwy koszt ruchu, przez co algorytm znajdzie najlepszą ścieżkę.
Na wejściu funkcja pobiera współrzędne pola startowego (`x1` i `y1`) oraz pola końcowego (`x2` i `y2`). Na wyjściu funkcja zwraca szacowany koszt potrzebny do przejścia od punktu startowego do końcowego.

```
1. function heuristic (x1, y1, x2, y2) {
2.   var distance, abs = Math.abs, max = Math.max,
3.     dx = x2 – x1,
4.     dy = y2 – y1,
5.     dd = dy – dx;
6.   distance = max(abs(dd), max(abs(dx), abs(dy)));
7.   return distance * min_koszt;
8. }
```

Listing 4. Implementacja funkcji heurystycznej Manhattan

Funkcja przeszacowana różni się od niedoszacowanej jedynie wartością, przez którą mnoży się dystans – zamiast najniższego występującego na mapie kosztu ruchu (najniższej wagi krawędzi w grafie) użyty jest koszt najwyższy:

```
7.   return distance * max_koszt;
```

Listing 5. Zastosowanie najwyższego kosztu w funkcji heurystycznej Manhattan

Przeszacowanie znacznie skraca czas obliczeń (algorytm najczęściej wybiera pierwsze pole, na które trafi, jako to najtańsze). Odbija się to jednak na dokładności – bardzo często będzie istnieć przynajmniej jedna ścieżka, której koszt okaże się niższy niż odnaleziony przez algorytm z taką heurystyką.

4.4. Elementy wspólne

Poniżej opisano elementy używane przez każdy z algorytmów. Pierwsza funkcja (listing 6.) odpowiada za sprawdzenie dostępności pola. Za parametry wejściowe przyjmuje współrzędne `x` i `y` pola, które ma zostać sprawdzone. Na wyjściu funkcja zwraca wartość logiczną (`true` lub `false`) oznaczającą dostępność pola. Pole jest dostępne, jeżeli istnieje na mapie oraz koszt wejścia na to pole jest mniejszy niż 999 punktów (wartość symboliczna przyjęta przez Autorów pracy). Funkcja wykorzystuje również zmienną globalną `worldMap`, w której przechowywane są informację o planszy.

```
1. function accessibility (x, y) {
2.    var map = worldMap;
3.    if (typeof map[x] === ‘undefined’){
4.       return false;
5.    }
6.    if (typeof map[x][y] === ‘undefined’){
7.       return false;
8.    }
9.    if (map[x][y].cost >= 999){
10.       return false;
11.    }
12.    return true;
13. }
```

Listing 6. Funkcja accessibility

Funkcja przedstawiona na listingu 7. jest odpowiedzialna za konwersję wyniku zwróconego przez algorytm i podanie poprawnej ścieżki (zmienna `path`). W tym celu funkcja pobiera identyfikator pola startowego (`start`) i końcowego (`stop`), identyfikator następnego pola (`next`) oraz tablicę relacji pomiędzy polami (`P`). Zmienna `path` odpowiada za przechowywanie ścieżki. Algorytm wykonuje się rekurencyjnie, krok po kroku uzupełniając zmienną path o kolejne węzły. Funkcja przerywa swoje działanie, gdy aktualny węzeł jest polem końcowym.

```
1. function parsePath (start, stop, next, P, path) {
2.   path.push(next);
3.   if (next === stop) {
```

```
4.    return path;
5.  } else {
6.    next = P[next];
7.    return parsePath(start, stop, next, P, path);
8.  }
9. }
```

Listing 7. Funkcja parsePath

W listingu 8. przedstawiono blok kodu odpowiedzialny za przygotowanie ścieżki do graficznego przedstawienia jej na mapie. Na początku deklarowane są zmienne, które zostaną przekazane do opisanej wcześniej funkcji `parsePath`, oraz zmienne pomocnicze dla pętli. Warto zauważyć, że funkcja `parsePath` wędruje od pola końcowego do początkowego – takie działanie jest wymuszone sposobem tworzenia relacji pomiędzy węzłami. W każdym kroku pętli do zmiennej `path` dodawane są wartości `x`, `y` oraz `cost`, które zostaną potem wykorzystane przy renderowaniu ścieżki w module gry.

```
1.  var start_id = map[end_x][end_y].id,
2.      stop_id = map[start_x][start_y].id,
3.      temp = parsePath(start_id, stop_id, start_id, P, []),
4.      path = {}, counter;
5.
6.  for (counter = 0; counter < temp.length; counter++) {
7.    i = temp[counter];
8.    if (typeof M[i] !== 'undefined') {
9.      path[(counter+1)] = [];
10.       path[(counter+1)][1] = M[i].x;
11.       path[(counter+1)][2] = M[i].y;
12.       path[(counter+1)][3] = M[i].cost;
13.   }
14. }
```

Listing 8. Blok przygotowujący ścieżkę do przedstawienia na mapie

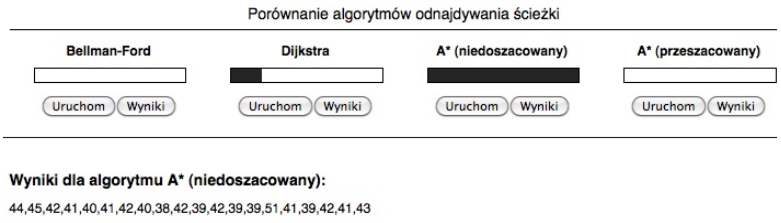
5. Analiza algorytmów

W dzisiejszych czasach zróżnicowanie sprzętu komputerowego jest duże, dlatego też testy zostały przeprowadzone w szerszej grupie kontrolnej: w pięćdziesięciu różnych środowiskach sprzętowych, w których każdy z testów powtórzono dwudziestokrotnie.

5.1. Aplikacja do testów

Specjalnie na potrzeby testów została napisana aplikacja umożliwiająca w bardzo prosty sposób przeprowadzenie badań przez osoby trzecie. Na rys. 8. przedstawiono jej interfejs.

Wszystkie algorytmy miały zakodowany punkt startowy oraz końcowy wyszukiwanej trasy, dzięki czemu każdorazowe uruchomienie funkcji implementującej dany algorytm odbywało się z wykorzystaniem identycznych danych wejściowych. W ten sposób uniknięto ewentualnego zafałszowania wyniku poprzez wyeliminowanie czynników losowych innych niż specyfikacja maszyny testowej.

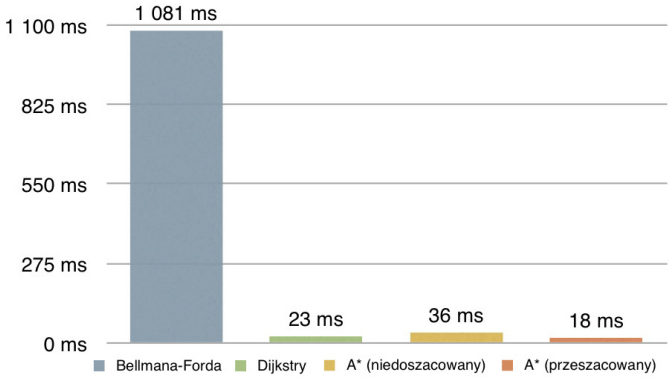


Rys. 8. Interfejs graficzny aplikacji testowej

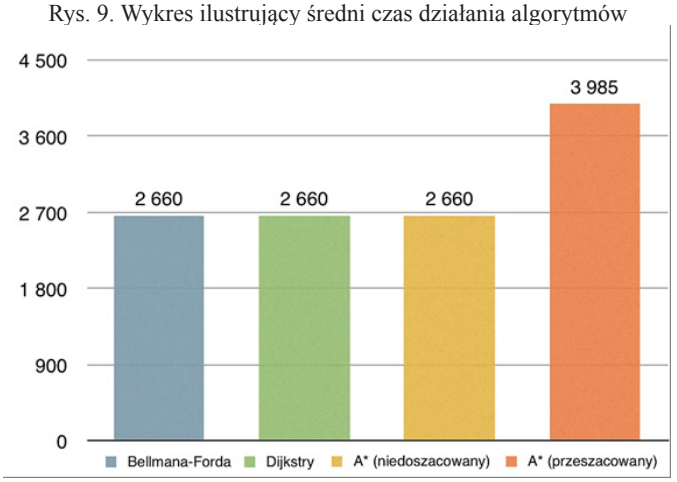
5.2. Wyniki testów

Pierwszym z badanych kryteriów był czas potrzebny do odnalezienia najkrótszej ścieżki przez każdy z programów, mierzony od chwili startu do momentu odnalezienia rozwiązania. Wynik testu został przedstawiony na rysunku 9. Jak widać, najszybszy okazał się algorytm A* (przeszacowany), a następnie algorytm Dijkstry.

Drugim kryterium wyboru był koszt przejścia (suma kosztów wejścia na każde z pól) dla wytyczonych tras. Im niższy koszt znalezionej trasy, tym lepiej. Na rys 10. przedstawiono wyniki dla każdego z algorytmów. Jak można zauważyć, prawie wszystkie algorytmy odnalazły taką samą trasę.



Rys. 9. Wykres ilustrujący średni czas działania algorytmów



Rys. 10. Wykres ilustrujący koszt znalezionych ścieżek

Z przeprowadzonych badań wynika, że najszybszym algorytmem był algorytm A* (przeszacowany), jednak jednocześnie okazał się najmniej dokładny, jeśli chodzi o koszt znalezionej trasy.

Tabela 2. zawiera podsumowanie wszystkich testów. W kolumnie „Punkty" przedstawiono iloczyn czasu i kosztów. Im niższy wynik, tym lepszy algorytm.

Tab. 2. Podsumowanie wyników testów

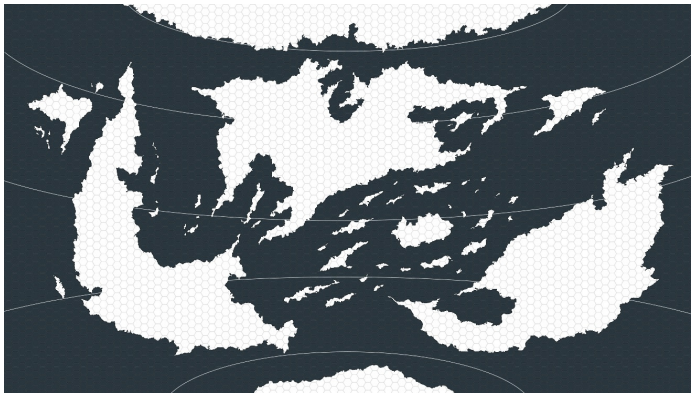
	Średni czas obliczeń	Koszt przejścia	Punkty
Bellmana-Forda	1081 ms	2660	2 875 460
Dijkstry	23 ms	2660	61 180
A* (niedoszacowany)	36 ms	2660	95 760
A* (przeszacowany)	18 ms	3985	71 730

Najlepszym rozwiązaniem dla gier przeglądarkowych wykorzystujących technologię JavaScript okazał się algorytm Dijkstry, który uzyskał wynik lepszy o około 17% od przeszacowanego algorytmu A*, o około 56% od niedoszacowanego i aż o 470% od algorytmu Bellmana-Forda.

6. Wdrożenie najlepszego rozwiązania problemu

6.1. Projekt systemu

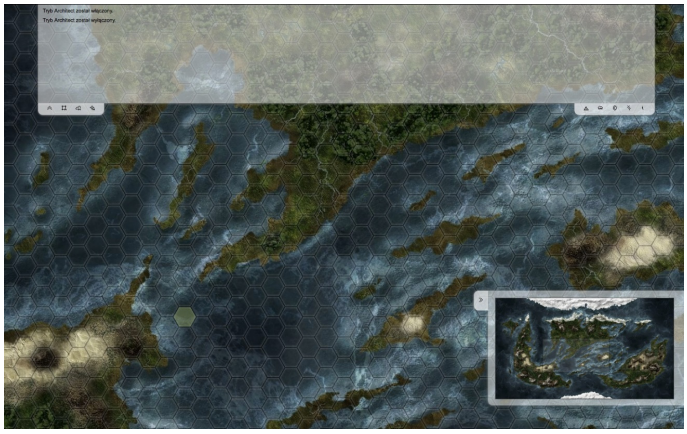
Z rezultatów przeprowadzonych badań wynika, że optymalnym rozwiązaniem jest algorytm Dijkstry. W tym rozdziale zostanie zaprezentowane wykorzystanie tego skryptu w formie rozbudowanej i przystosowanej do współpracy z planszą gry przedstawiającą fikcyjny świat.
Plansza została zbudowana z 4900 sześciokątnych pól ułożonych w siatkę o wymiarach 49 × 100 pól. Pola mogą przedstawiać jeden z trzynastu typów terenu – każdy o innym koszcie ruchu, który symbolizuje trudność poruszania się po danym terenie (im bardziej wymagający, tym wyższy koszt pola).
Dla zróżnicowania kosztów oraz zwiększenia realizmu oprócz typów terenu zastosowano system pogodowy. Pogoda modyfikuje koszt terenu o pewną symboliczną wartość i tym samym przyczynia się do ułatwienia bądź utrudnienia podróży.
Głównym założeniem przy projektowaniu systemu pogodowego było stworzenie silnika prostego do implementacji, który jednocześnie realistycznie odwzorowywałby występowanie zjawisk atmosferycznych.
W celu osiągnięcia zamierzonego efektu planszę podzielono na sześć stref klimatycznych (rys. 11). Przynależność pola do danej strefy wylicza się na podstawie współrzędnych. W systemie funkcjonuje osiem typów pogody. Każdy z nich posiada modyfikator prawdopodobieństwa wystąpienia w zależności od typu terenu oraz strefy klimatycznej, do której należy pole.



Rys. 11 Podział na strefy klimatyczne

6.2. Aplikacja do zarządzania danymi

Na potrzeby projektu napisano aplikację, która ułatwia zarządzanie danymi. Udostępnia ona trzy tryby pracy: wizualizację danych, edycję terenu oraz generator pogody. Na rys.12. przedstawiono podgląd interfejsu aplikacji.



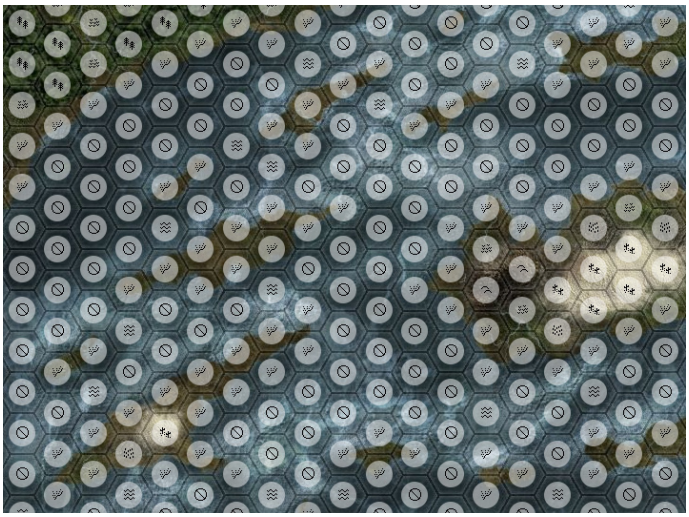
Rys. 12 Ogólny widok interfejsu

Aplikacja pracuje w kilku trybach:

- wizualizacji danych – tryb umożliwia podgląd różnych właściwości pól: typu terenu, pogody panującej na danym polu, kosztu ruchu (składającego się z kosztu terenu i modyfikatora pogodowego) oraz współrzędnych pól.
- edycji terenu – w ramach niego udostępniono użytkownikowi możliwość grupowego zmieniania typu terenu.
- generowania pogody – umożliwia automatyczne generowanie typów pogody zgodnie z założeniami systemu pogodowego

Na rys. 13. przedstawiono fragment widoku z informacjami na temat typu terenu (dla zwiększenia czytelności oznaczenie typów terenów znajduje się na rys. 14.). Użytkownik może przełączać się pomiędzy widokami poprzez wybranie odpowiedniej opcji z przybornika narzędzi (rys. 15.). Szczegółowy opis obsługi poszczególnych trybów aplikacji znajduje się w pracy *Analiza i implementacja* […][11].

11. T. Sobczak, *op. cit.*



Rys. 13. Przykładowy widok z informacjami na temat typu terenu

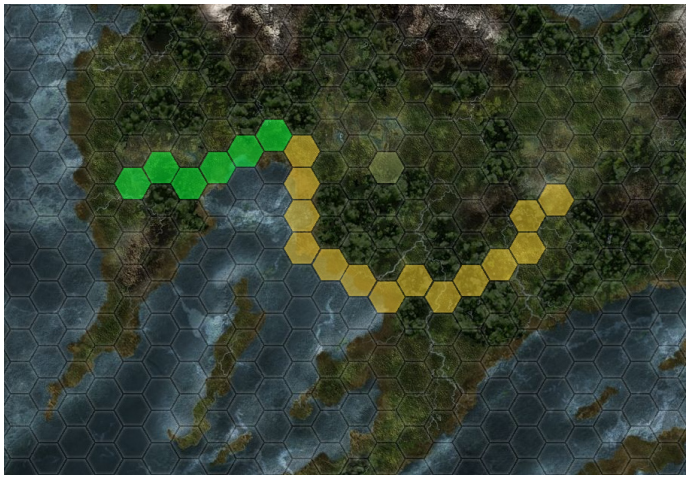
Rys. 14. Oznaczenie typów terenu

Rys. 15. Przybornik narzędzi dla trybu wizualizacji

6.3. Budowa modułu nawigacji

Skrypt symulujący przemieszczanie się gracza po planszy również został włączony do aplikacji z narzędziami jako dodatkowy tryb. Po jego wybraniu wystarczy wskazać początek i koniec trasy, a następnie wcisnąć przycisk start (). Po chwili program narysuje nam wyliczoną trasę, po czym rozpocznie jej animowanie.

Na rys. 16. zaprezentowano przykładową trasę w trakcie animacji. Kolorem żółtym zaznaczono wybraną trasę, zielonym zaś pola, które zostały już oznaczone przez system jako przebyte. Czas przejścia pomiędzy polami wyliczany jest na podstawie kosztu pola docelowego pomnożonego przez jedną dziesiątą sekundy.



Rys. 16. Przykładowa trasa w trakcie animacji

7. Podsumowanie

W artykule przedstawiono trzy najbardziej popularne algorytmy odnajdywania ścieżek. Przeprowadzono analizę porównawczą, która polegała na zestawieniu ze sobą średnich czasów działania oraz zwróconych kosztów tras znalezionych przez wszystkie algorytmy. Na tej podstawie wybrano algorytm Dijkstry, który na co dzień jest bardzo często wykorzystywany przy wyznaczaniu tras dla pakietów danych w sieciach komputerowych (np. w protokole OSPF).

Autorskim projektem jest moduł gry przeglądarkowej odpowiedzialny za nawigację, który wykorzystuje implementację algorytmu Dijkstry do odnajdywania trasy na mapie. Jego główną zaletą jest zróżnicowanie kosztów przejścia pomiędzy polami wynikające z zaimplementowanego systemu typów terenu i pogody. Najważniejszą wadą tego modułu może być sposób uruchomienia procedury nawigacyjnej, który obecnie jest mało realistyczny (trzeba samodzielnie wyznaczyć punkt startowy – w standardowej rozgrywce gracz powinien startować z miejsca, w którym się aktualnie znajduje). Oprócz modułu nawigacyjnego napisano narzędzia ułatwiające zarządzanie modyfikatorami pól planszy.

Projekt w przyszłości można rozwinąć poprzez zwiększenie realizmu podróży, dodając modyfikatory przejść pomiędzy polami – z niektórych terenów łatwiej jest wyjść niż do nich wejść (np. góry, morze). Dodatkowo można przeprowadzić szerszą analizę algorytmu A* w celu odnalezienia optymalnej funkcji heurystycznej, która sprawi, że czas działania algorytmu i zwrócony koszt odnalezionej trasy będzie lepszy niż obecne wyniki algorytmu Dijkstry.

Bibliografia

1. Ross K. A., Wright C. R. B., *Matematyka dyskretna*, Wydawnictwo Naukowe PWN, Warszawa 1996.

2. Sysło M. M., Deo N., Kowalik J. S., *Algorytmy optymalizacji dyskretnej*, Wydawnictwo Naukowe PWN, Warszawa 1995.

3. T. Sobczak, *Analiza i implementacja algorytmów odnajdywania ścieżki do zastosowania w grach przeglądarkowych,* inżynierska praca dyplomowa, Wrocławska Wyższa Szkoła Informatyki Stosowanej, Wrocław 2012.

4. *Algorytmy i struktury danych*, wykład 7: http://hector.tu.kielce.pl/przedmioty/aisd-wyk/AiSD_w07.pdf, dostępny w internecie [dostęp 30.11.2021]