Przetwarzanie i przechowywanie opisu siatki trójkątnej na płaszczyźnie

Mateusz Zając, Błażej Kapkowski

Opis programu

Program daje możliwość wczytywania i zapisywania opisów siatek trójkątnych. Umożliwia wyświetlanie tychże siatek. Pozwala na wykonanie trzech operacji na siatkach, korzystając z jednej z dwóch struktur danych - list wierzchołków i trójkątów oraz z Half Edge Data Structure. Pierwsza operacja to przeglądanie sąsiednich wierzchołków względem wybranego wierzchołka, kolejna to przeglądanie sąsiednich trójkątów względem trójkąta, ostatnia to przeszukiwanie kolejnych sąsiednich trójkątów w poszukiwaniu wierzchołka. Aplikacja ma formę notatnika Jupytera, zawierającego pewne funkcje, które użytkownik może wywoływać we własnych komórkach. Do wizualizacji wykorzystano narzędzie napisane przez KN BIT.

Wymagania techniczne

- Python 3.10.12
- biblioteki:
    - numpy
    - matplotlib

Wersje bibliotek podane są w pliku `requirements.txt`

Jak używać

By skorzystać z programu, należy otworzyć notatnik `projekt.ipynb`, a potem na jego końcu stworzyć komórkę i w niej wywołać odpowiednią funkcję.

W programie często wykorzystywanymi nazwami argumentów są `vertices` i `triangles` (zamiennie z `faces`). `vertices` to lista wierzchołków, czyli krotek dwóch liczb rzeczywistych (współrzędnych), a `triangles` i `faces` to lista trójkątów, czyli krotek trzech liczb całkowitych (indeksów wierzchołków trójkąta, numerowanych od 0).

Wczytywanie z pliku, zapisywanie do niego

Triangulacje wczytuje się za pomocą funkcji `load_from_file(filename)` gdzie `filename` jest nazwą pliku znajdującego się w katalogu `triangulations`. Może to być np. `sample_triangulation4.txt`. Funkcja zwraca krotkę - listę wierzchołków i listę trójkątów.

Triangulację można zapisać do pliku za pomocą funkcji `save_to_file(filename, vertices, triangles)`, gdzie `filename` to nazwa pliku który pojawi się w katalogu `triangulations`.

Każdy wiersz pliku z triangulacją przyjmuje jedną z dwóch postaci:

- litera `v` i dwie liczby rzeczywiste, wszystko oddzielone spacją, np. `v 1.0 2.0`,
- litera `t` i trzy liczby całkowite, wszystko oddzielone spacją, np. `t 0 1 4`.

Wyświetlenie triangulacji:

By wyświetlić siatkę, należy wywołać funkcję `draw_triangulation(vertices, triangles, vis)`, gdzie `vis` to obiekt klasy `Visualizer()`. Funkcja zwraca obiekt tej samej klasy, co `vis`. Należy na nim

użyć metody `.show()`.

```
vis = Visualizer()
vertices, triangles = load_from_file("sample_triangulation4.txt")
vis = draw_triangulation(vertices, triangles, vis)
vis.show()
save_to_file("triangulation_copy.txt", vertices, triangles)
```

Operacje na listach wierzchołków i trójkątów

Pierwsza operacja

Żeby przeszukać siatkę w poszukiwaniu wierzchołków sąsiadujących z innym wierzchołkiem, trzeba użyć funkcji `find_neighbors(vertex_index, triangles, second_level)`. `vertex_index` to indeks wierzchołka, którego sąsiadów ma znaleźć funkcja. `second_level` to argument typu bool. Ustawienie go na True zleca funkcji przeszukanie dwóch kolejnych warstw sąsiadów. False - tylko jednej. Funkcja zwraca listę składającą się z 2 słowników (pythonowy set()) z indeksami sąsiadów. Pierwszy element listy to wierzchołki z pierwszej warstwy, a drugi z drugiej, jeśli te znaleziono (w przeciwnym razie, ten zbiór jest pusty).

W celu wizualizacji tej operacji, należy wywołać funkcję `find_neighbors_visualize(vertex_index, vertices, triangles, second_level)`. Zwraca ona krotkę: listę 2 słowników jak powyżej oraz obiekt klasy `Visualizer()`. Wywołując na nim metodę `.show()` można wyświetlić wizualizację algorytmu, jako obraz.

```
vertices, triangles = load_from_file("sample_triangulation4.txt")
neighbors, vis = find_neighbors_visualize(5, vertices, triangles, True)
vis.show()
```

Druga operacja

Przeszukiwanie trójkątów sąsiadujących z wybranym trójkątem wykonuje się za pomocą funkcji `find_triangle_neighbors(selected_index, triangles, second_level)`. `selected_index` to indeks trójkąta, którego sąsiadów się szuka. `second_level` pełni podobną rolę, co w pierwszej operacji - to wartość bool, która decyduje ile warstw ma być znalezionych. Funkcja zwraca listę dwóch zbiorów. Pierwszy zbiór zawiera indeksy trójkątów z pierwszej warstwy, drugi z drugiej.

Wizualizacja: wywołanie funkcji `find_triangle_neighbors_visualize(selected_index, vertices, triangles, second_level)`. Zwraca listę zbiorów, jak poprzednio, a także obiekt `Visualizer()`.

```
vertices, triangles = load_from_file("sample_triangulation4.txt")
neighbors, vis = find_triangle_neighbors_visualize(2, vertices, triangles, True)
vis.show()
```

Trzecia operacja

Żeby przeszukać siatkę w poszukiwaniu wybranego wierzchołka, iterując po kolejnych, sąsiednich trójkątach, należy wywołać funkcję `find_triangle_containing_point(start_triangle, point, triangles)`, gdzie `start_triangle` to indeks trójkąta od którego się zaczyna iterację, `point` - indeks wierzchołka który się szuka. Funkcja zwraca trójkąt - czyli krotkę indeksów wierzchołków - który zawiera wskazany wierzchołek. Jeżeli jest w stanie go znaleźć, bo w przeciwnym razie zwraca None.

Żeby zwizualizować tą operację, trzeba wywołać `find_triangle_containing_point_visualize(start_triangle, point, vertices, triangles, vis)`, gdzie `vis` to wcześniej zainicjalizowany obiekt `Visualizer()`. Następnie można wykonać `vis.show()`, by wyświetlić wizualizację.

```
vis = Visualizer()
vertices, triangles = load_from_file("sample_triangulation4.txt")
```

```
triangle_found = find_triangle_containing_point_visualize(8, 2, vertices, triangles, vis)
vis.show()
```

Konstrukcja i wykorzystanie struktury Half Edge

By utworzyć strukturę Half Edge, należy zainicjalizować klasę `HalfEdgeGraph(vertices, faces)`.

Pierwsza operacja

Operacja przeszukiwania sąsiednich wierzchołków wykonywana jest przez metodę `.incidental_vertices(vertex_index, levels)`, gdzie `vertex_index` to wierzchołek z którego się zaczyna, a `levels` to liczba warstw. Metoda zwraca listę słowników, gdzie każdy słownik odpowiada kolejnej warstwie i każdy zawiera indeksy wierzchołków z danej warstwy.

Żeby zwizualizować operację, trzeba wykonać `.incidental_vertices_visualize(vertex_index, levels, visualizer)`, gdzie `visualizer` to wcześniej zainicjalizowany obiekt `Visualizer()`.

Zamiast metody `.show()`, można zastosować metodę `.show_gif(delay)`, która przedstawi kolejne kroki wykonywania algorytmu, każdy krok pokazany co `delay` sekund.

```
vis = Visualizer()
vertices, triangles = load_from_file("sample_triangulation4.txt")
vis = draw_triangulation(vertices, triangles, vis)
half_edges = HalfEdgeGraph(vertices, triangles)
neighbors, vis = half_edges.incidental_vertices_visualize(5, 2, vis)
vis.show_gif(500)
```

Druga operacja

Przeglądanie sąsiednich trójkątów - używa się metody `.incidental_triangles(face_index, levels)`. `face_index` - indeks startowego trójkąta, `levels` - liczba warstw. Funkcja zwraca listę słowników, każdy odpowiada kolejnej warstwie i zawiera indeksy trójkątów z tej warstwy.

Wizualizacja - metoda `.incidental_triangles_visualize(face_index, levels, visualizer)`. Sposób wywoływania analogiczny jak w pierwszej operacji.

```
vis = Visualizer()
vertices, triangles = load_from_file("sample_triangulation4.txt")
vis = draw_triangulation(vertices, triangles, vis)
half_edges = HalfEdgeGraph(vertices, triangles)
neighbors, vis = half_edges.incidental_triangles_visualize(2, 2, vis)
vis.show_gif(500)
```

Trzecia operacja

Przeszukiwanie trójkąta zawierającego dany wierzchołek - należy użyć metody `.seek_vertex(face_index, vertex_index)`. `face_index` - początkowy trójkąt, `vertex_index` - szukany wierzchołek. Metoda zwraca indeks znalezionego trójkąta, jeśli taki istnieje. Jeśli nie istnieje, zwraca None.

Wizualizacja - metoda `.seek_vertex_visualize(face_index, vertex_index, visualizer)`. Wywoływanie - analogiczne.

```
vis = Visualizer()
vertices, triangles = load_from_file("sample_triangulation4.txt")
vis = draw_triangulation(vertices, triangles, vis)
half_edges = HalfEdgeGraph(vertices, triangles)
found_triangle, vis = half_edges.seek_vertex_visualize(2, 2, vis)
vis.show_gif(500)
```

Przetwarzanie i przechowywanie opisu siatki trójkątnej na płaszczyźnie

Mateusz Zając, Błażej Kapkowski

Opis problemu

Zadanie polegało na porównaniu wydajności dwóch struktur służących do przechowywania opisu siatki trójkątnej na płaszczyźnie. Te struktury to:

- lista współrzędnych wierzchołków i lista trójkątów – czyli trójek indeksów wierchołków, które tworzą dany trójkąt,
- Half Edge data structure.

Na obu strukturach wykonano trzy operacje:

- wyznaczenie otoczenia dla wybranego wierzchołka,
- wyznaczenie otoczenia dla wybranego trójkąta,
- przeglądanie sąsiednich trójkątów w celu znalezienia trójkąta zawierającego dany wierzchołek.

Porównano ich wydajność obliczeniową i pamięciową.

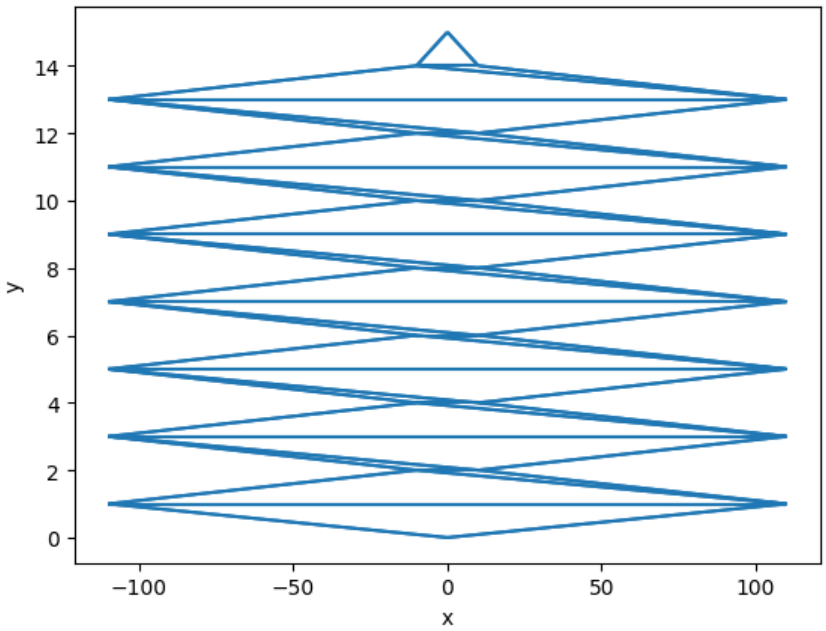
Dane techniczne

W eksperymencie skorzystano z następujących narzędzi:

- Język programowania: Python 3.10.12
- System operacyjny: Linux Mint 21 x86 64 (kernel: 5.15.0-91-generic)
- CPU: 8-rdzeniowy Intel i7-6700HQ 3.500GHz
- GPU: Intel HD Graphics 530 + NVIDIA GeForce GTX 960M
- RAM: 16 GB
- do wizualizacji wykorzystano program napisany przez KN BIT
- zewnętrzne biblioteki Pythona: memory-profiler, numpy, pandas, matplotlib

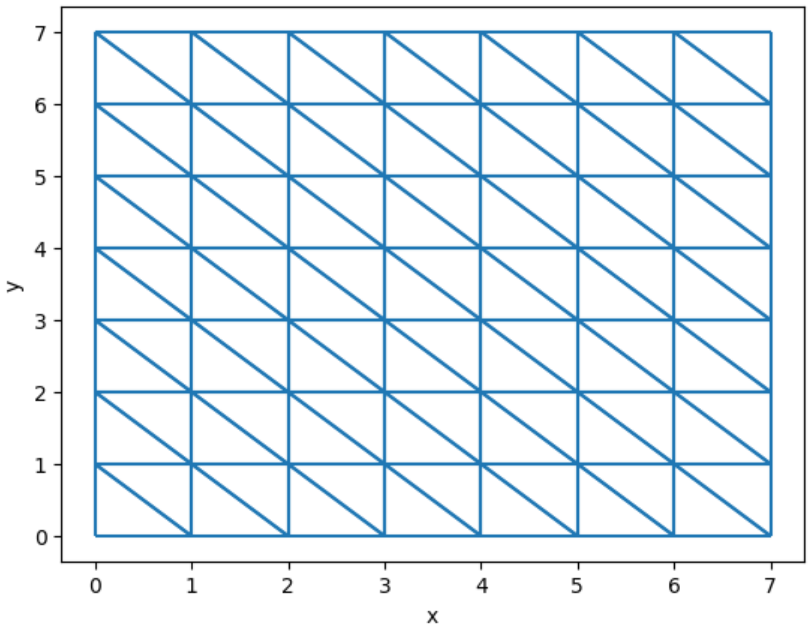
Rodzaje testów

Testy wykonano na dwóch rodzajach siatek. Wybrano je tak, by można było łatwo generować siatki o zadanej wielkości. Na potrzeby sprawozdania, nazwano je Siatkami typu 1 i Siatkami typu 2. Siatki typu 1 zbudowane są z symetrycznych lub prawie symetrycznych wielokątów, które mają po (2n – 2) / 2 wierzchołków na prawo i na lewo od osi symetrii, a także po 1 punkcie na szczycie i na dole. Wielokąty te poddano triangulacji za pomocą funkcji napisanej na potrzeby wcześniejszych laboratoriów.



Ilustracja 1: wizualizacja pierwszego typu siatek

Siatki typu 2 powstały z podzielenia każdego pola siatki kwadratowej (o polach z bokami o długości 1) na dwa trójkąty. Ten typ siatek zastosowano, bo średnia ilość wierzchołków i trójkątów w otoczeniu, dla wyższych warstw, będzie większa, tak więc obliczenia powinny trwać dłużej. Ponadto, wygenerowanie takich siatek zajmuje mniej czasu, nawet gdy są duże – bo niepotrzebne jest obliczanie triangulacji.



Ilustracja 2: wizualizacja drugiego typu siatek

Wydajność obliczeniowa – siatki typu 1

Porównano wydajność obliczeniową obu struktur na pierwszym typie siatek. W ramach przekroju, przedstawiono wyniki testów dla każdej z trzech operacji. Dla pierwszej i drugiej operacji, przedstawiono wyniki testów dla szukanej 1 warstwy wierzchołków/trójkątów, a także dla 2 warstw. Wartości w tabelach podane są w sekundach.

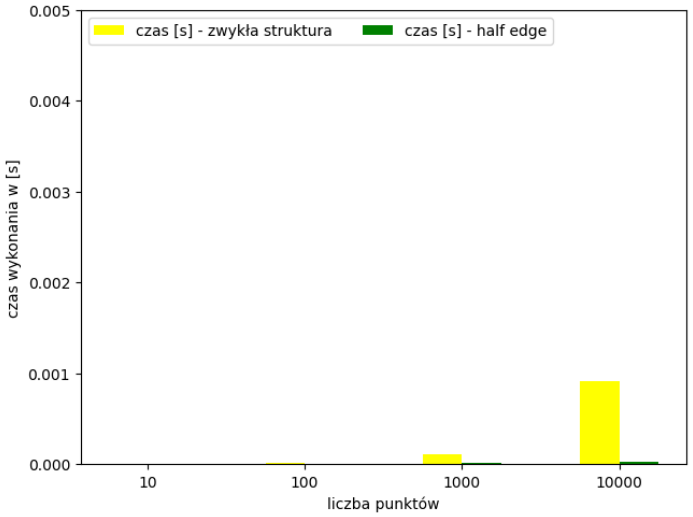


Diagram 1: pomiar czasu wykonania 1. operacji, 1 warstwa sąsiednich wierzchołków

	czas [s] - zwykła struktura	czas [s] - half edge	liczba wierzchołków
0	0.00001	0.00001	10
1	0.00002	0.00001	100
2	0.00011	0.00001	1000
3	0.00092	0.00002	10000

Tabela 1: pomiar czasu wykonania 1. operacji, 1 warstwa sąsiednich wierzchołków

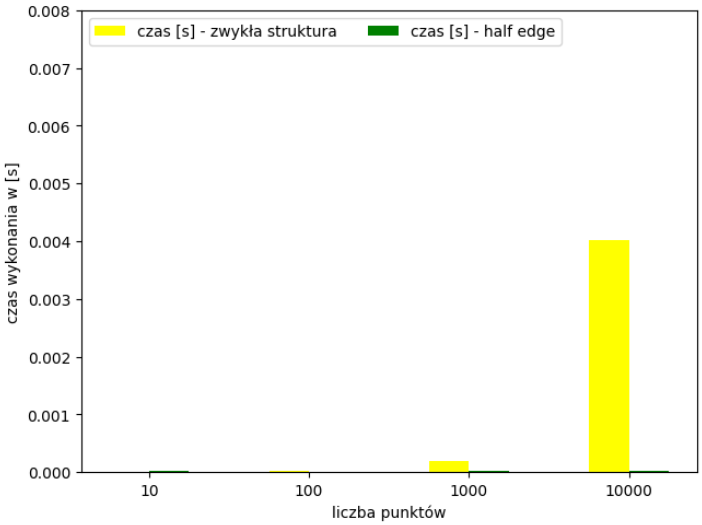


Diagram 2: pomiar czasu wykonania 1. operacji, 2 warstwy sąsiednich wierzchołków

	czas [s] - zwykła struktura	czas [s] - half edge	liczba wierzchołków
0	0.00001	0.00002	10
1	0.00003	0.00001	100
2	0.00019	0.00003	1000
3	0.00402	0.00002	10000

Tabela 2: pomiar czasu wykonania 1. operacji, 2 warstwy sąsiednich wierzchołków

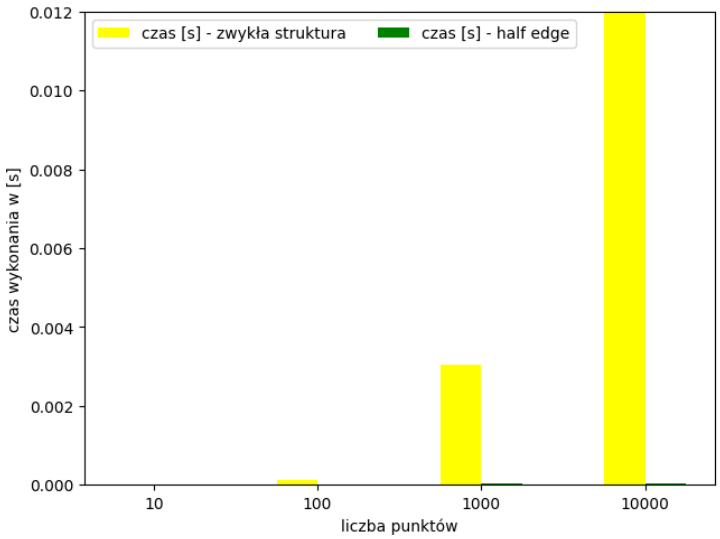


Diagram 3: pomiar czasu wykonania 2. operacji, 1 warstwa sąsiednich trójkątów

	czas [s] - zwykła struktura	czas [s] - half edge	liczba wierzchołków
0	0.00002	0.00001	10
1	0.00012	0.00001	100
2	0.00304	0.00003	1000
3	0.01880	0.00002	10000

Tabela 3: pomiar czasu wykonania 2. operacji, 1 warstwa sąsiednich trójkątów

Diagram 4: pomiar czasu wykonania 2. operacji, 2 warstwy sąsiednich trójkątów

	czas [s] - zwykła struktura	czas [s] - half edge	liczba wierzchołków
0	0.00004	0.00005	10
1	0.00046	0.00004	100
2	0.00592	0.00004	1000
3	0.03446	0.00002	10000

Tabela 4: pomiar czasu wykonania 2. operacji, 2 warstwy sąsiednich trójkątów

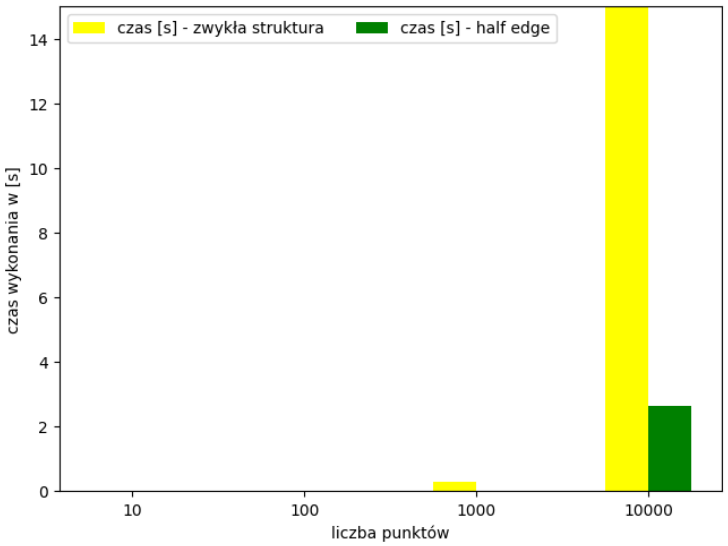


Diagram 5: pomiar czasu wykonania 3. operacji, przeszukiwanie trójkątów w poszukiwaniu wierzchołka

	czas [s] - zwykła struktura	czas [s] - half edge	liczba wierzchołków
0	0.00002	0.00001	10
1	0.00636	0.00008	100
2	0.29234	0.01549	1000
3	16.55994	2.63428	10000

Tabela 5: pomiar czasu wykonania 3. operacji, przeszukiwanie trójkątów w poszukiwaniu wierzchołka

Wyniki pomiarów przedstawione na powyższych ilustracjach (Diagramy i Tabele 1-5) wskazały, że jeśli chodzi o szybkość obliczeń, struktura Half Edge sprawdza się dużo lepiej. Najbardziej uwypuklają to Diagramy 2-4, gdzie praktycznie nie widać słupków odpowiadających pomiarom czasu dla Half Edge. Jak pokazuje Diagram 5, różnica zaczyna mieć szczególne znaczenie przy siatkach złożonych w większej liczby punktów – różnica w czasach obliczeń potrafiła osiągnąć prawie 9 sekund.

Wydajność obliczeniowa – siatki typu 2

Przeprowadzono testy dla siatek o większej liczbie sąsiadujących ze sobą trójkątów. Po to, by uwypuklić różnicę między wydajnościami obu struktur. Wybrano te zbiory, dla których różnice były największe.

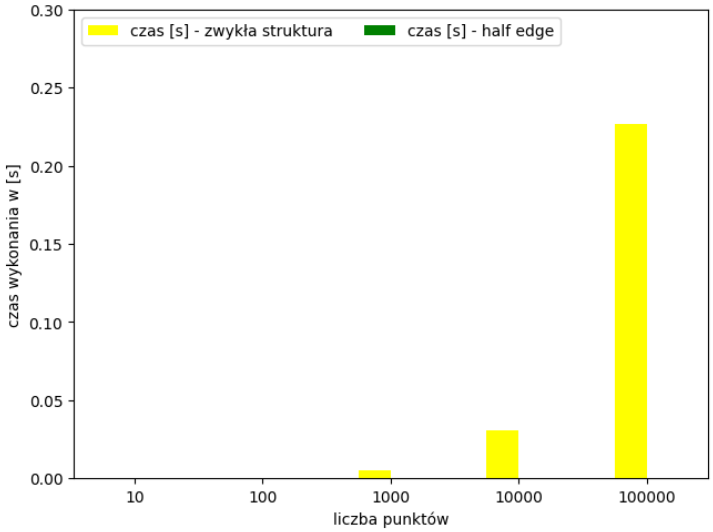


Diagram 6: pomiar czasu wykonania 2. operacji, 2 warstwy sąsiednich trójkątów

	czas [s] - zwykła struktura	czas [s] - half edge	liczba wierzchołków
0	0.00003	0.00001	10
1	0.00016	0.00002	100
2	0.00527	0.00001	1000
3	0.03080	0.00001	10000
4	0.22667	0.00003	100000

Tabela 6: pomiar czasu wykonania 2. operacji, 2 warstwy sąsiednich trójkątów

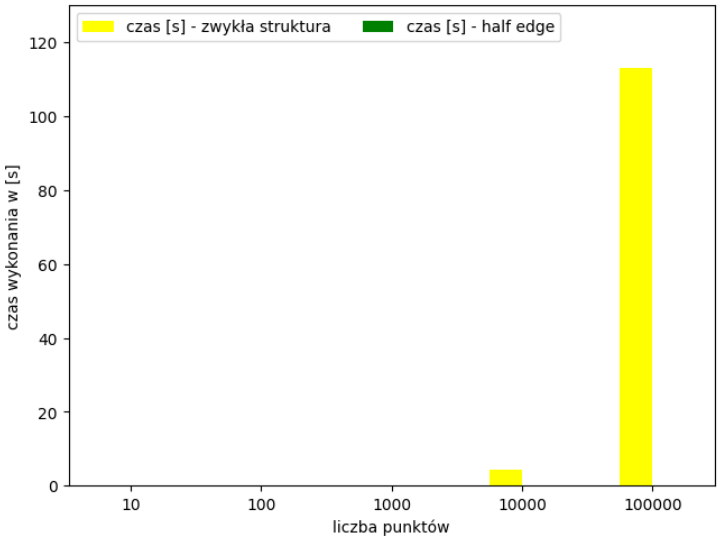


Diagram 7: pomiar czasu wykonania 3. operacji, przeszukiwanie trójkąta w poszukiwaniu wierzchołka

	czas [s] - zwykła struktura	czas [s] - half edge	liczba wierzchołków
0	0.00006	0.00001	10
1	0.00428	0.00002	100
2	0.13998	0.00002	1000
3	4.39678	0.00003	10000
4	113.10611	0.00004	100000

Tabela 7: pomiar czasu wykonania 3. operacji, przeszukiwanie trójkąta w poszukiwaniu wierzchołka

Różnice między wydajnościami operacji wzrosły tak, jak oczekiwano. W przypadku drugiej operacji (Tabela 6), dla największej liczby wierzchołków, czas obliczeń dla prostej struktury danych był około 7555 razy większy, niż dla Half Edge. Stosunek był jeszcze większy dla trzeciej operacji (Tabela 7). Dla 100 tysięcy wierzchołków, znalezienie trójkąta

zawierającego pewien wierzchołek zajęło ponad 113 sekund, a wykorzystanie Half Edge skróciło ten czas do ułamku sekundy.

Wydajność pamięciowa – siatki typu 1

Testy wykonano za pomocą biblioteki memory-profiler. Funkcja memory_usage() z tejże biblioteki czyta co pewien ułamek sekundy rozmiar pamięci wykorzystywanej w tej chwili. W każdej takiej próbce znajduje się pewna wartość maksymalna i minimalna. W wynikach przedstawiono ich różnice. Dla przekroju, przedstawiono wyniki wszystkich pomiarów.

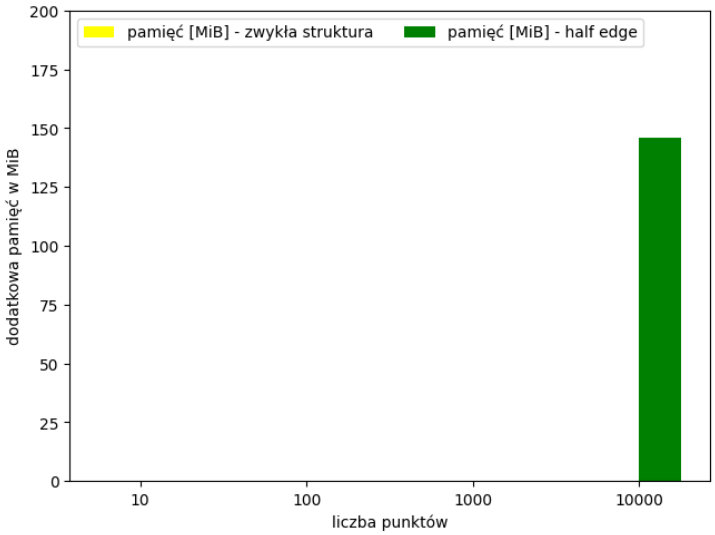


Diagram 8: pomiar pamięci używanej przez 1. operację, 1 warstwa sąsiednich wierzchołków

	pamięć [MiB] - zwykła struktura	pamięć [MiB] - half edge	liczba wierzchołków
0	0.00000	0.00000	10
1	0.00000	0.00000	100
2	0.00000	0.00000	1000
3	0.00000	145.84375	10000

Tabela 8: pomiar pamięci używanej przez 1. operację, 1 warstwa sąsiednich wierzchołków

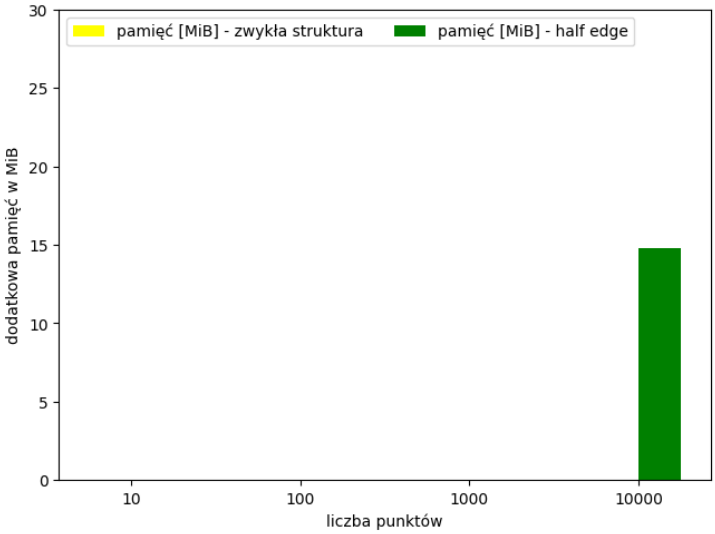


Diagram 9: pomiar pamięci używanej przez 1. operację, 2 warstwy sąsiednich wierzchołków

	pamięć [MiB] - zwykła struktura	pamięć [MiB] - half edge	liczba wierzchołków
0	0.00000	0.00000	10
1	0.00000	0.00000	100
2	0.00000	0.00000	1000
3	0.00000	14.78125	10000

Tabela 9: pomiar pamięci używanej przez 1. operację, 2 warstwy sąsiednich wierzchołków

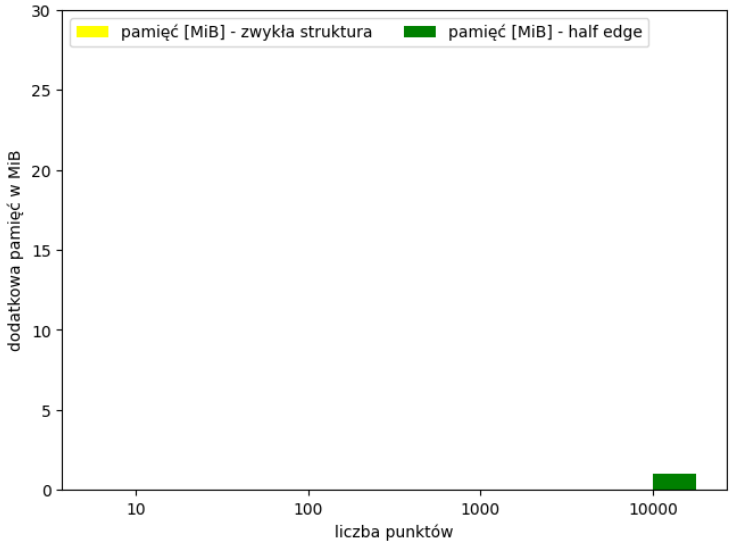


Diagram 10: pomiar pamięci używanej przez 2. operację, 1 warstwa sąsiednich trójkątów

	pamięć [MiB] - zwykła struktura	pamięć [MiB] - half edge	liczba wierzchołków
0	0.00000	0.00000	10
1	0.00000	0.00000	100
2	0.00000	0.01562	1000
3	0.00000	0.98438	10000

Tabela 10: pomiar pamięci używanej przez 2. operację, 1 warstwa sąsiednich trójkątów

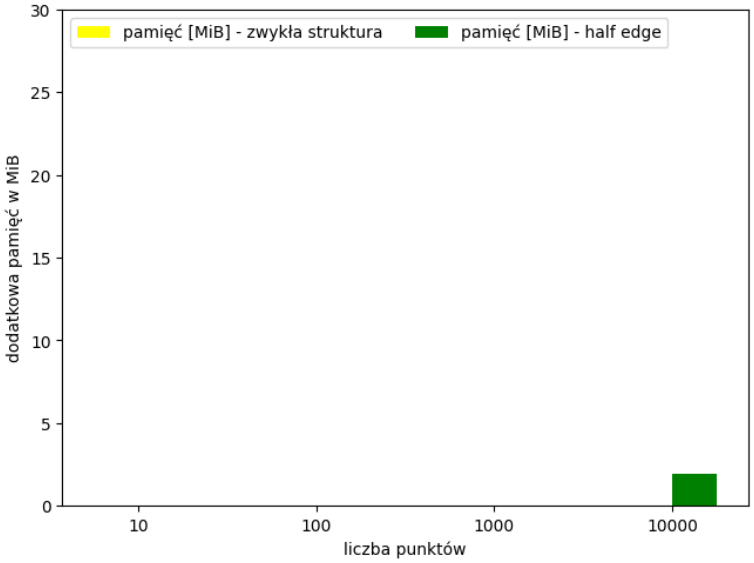
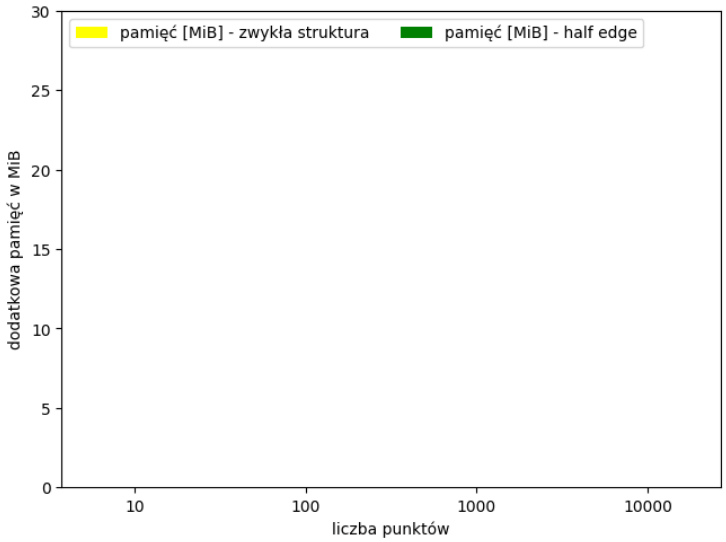


Diagram 11: pomiar pamięci używanej przez 2. operację, 2 warstwy sąsiednich trójkątów

	pamięć [MiB] - zwykła struktura	pamięć [MiB] - half edge	liczba wierzchołków
0	0.00000	0.00000	10
1	0.00000	0.00000	100
2	0.00000	0.00000	1000
3	0.00000	1.96875	10000

Tabela 11: pomiar pamięci używanej przez 2. operację, 2 warstwy sąsiednich trójkątów



	pamięć [MiB] - zwykła struktura	pamięć [MiB] - half edge	liczba wierzchołków
0	0.00000	0.00000	10
1	0.00000	0.00000	100
2	0.00000	0.00000	1000
3	0.00000	0.00000	10000

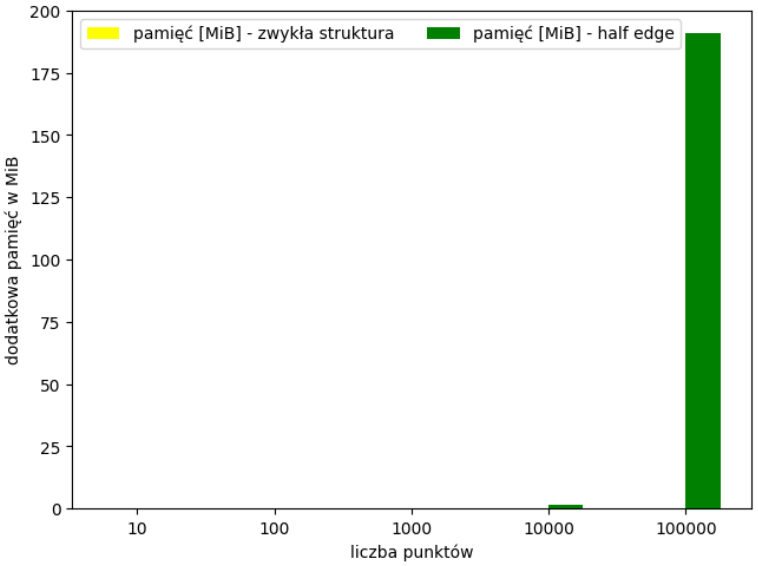
Diagram 12: pomiar pamięci używanej przez 3. operację, przeszukiwanie sąsiednich trójkątów w poszukiwaniu wierzchołka

Tabela 12: pomiar pamięci używanej przez 3. operację, przeszukiwanie sąsiednich trójkątów w poszukiwaniu wierzchołka

Średnio, przy większych siatkach, operacje na strukturze Half Edge zajmowały więcej pamięci. Pokazują to Diagramy i Tabele 1-4. W przypadku mniejszych siatek, różnica była zbyt mała, by zwróciła ją funkcja biblioteczna używana do mierzenia zajmowanej pamięci. Funkcja zwracała wartości w MiB.
Trudno powiedzieć, co mogło spowodować, że wartości w Tabeli 5 są równe zeru. Mogło to być spowodowane przeoczonym błędem, jednak w późniejszej części sprawozdania sprawdzono tą samą operację na większych siatkach. Wtedy ten artefakt się nie pojawił. Więc możliwe, że wpływ mogło mieć zarządzanie pamięci przez notatniki Jupytera. Mimo, że podjęto starań, by uniknąć takich sytuacji.
Podobnie dziwnym zjawiskiem jest, że pierwsza operacja, przy wyznaczaniu 1 warstwy, zajęła tak dużo pamięci (prawie 146 MiB).

Wydajność pamięciowa – siatki typu 2

Żeby uwypuklić różnice w wykorzystywanej pamięci, operacje przeprowadzono także na większych siatkach. Poza tym, jako że testy wykonane zostały po testach dla poprzedniego typu, artefakty pochodzące z alokowania przez notatnik pamięci były mniej prawdopodobne. Choć trudno to stwierdzić bez głębszej znajomości implementacji bibliotek.



	pamięć [MiB] - zwykła struktura	pamięć [MiB] - half edge	liczba wierzchołków
0	0.00000	0.00000	10
1	0.00000	0.00000	100
2	0.00000	0.00000	1000
3	0.00000	1.31250	10000
4	0.00000	190.72266	100000

Diagram 13: pomiar pamięci używanej przez 1. operację, 1 warstwa sąsiednich wierzchołków

Tabela 13: pomiar pamięci używanej przez 1. operację, 1 warstwa sąsiednich wierzchołków

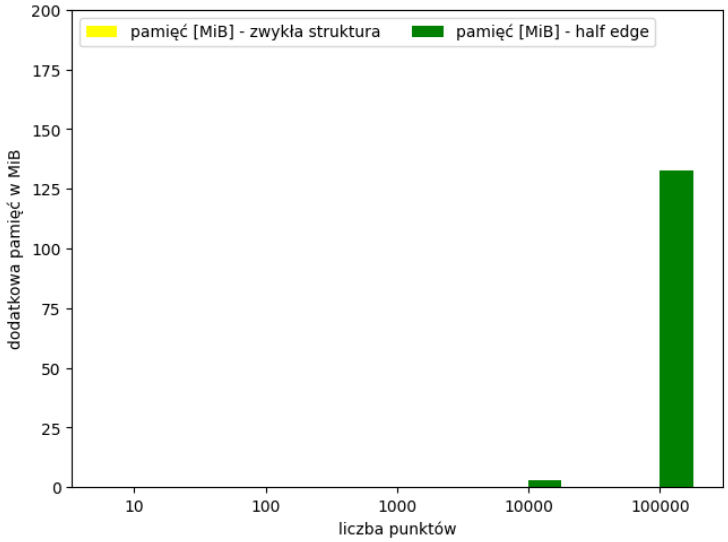


Diagram 14: pomiar pamięci używanej przez 2. operację, 1 warstwa sąsiednich trójkątów

	pamięć [MiB] - zwykła struktura	pamięć [MiB] - half edge	liczba wierzchołków
0	0.00000	0.00391	10
1	0.00000	0.00000	100
2	0.00000	0.00000	1000
3	0.00000	3.12500	10000
4	0.00000	132.49219	100000

Tabela 14: pomiar pamięci używanej przez 2. operację, 1 warstwa sąsiednich trójkątów

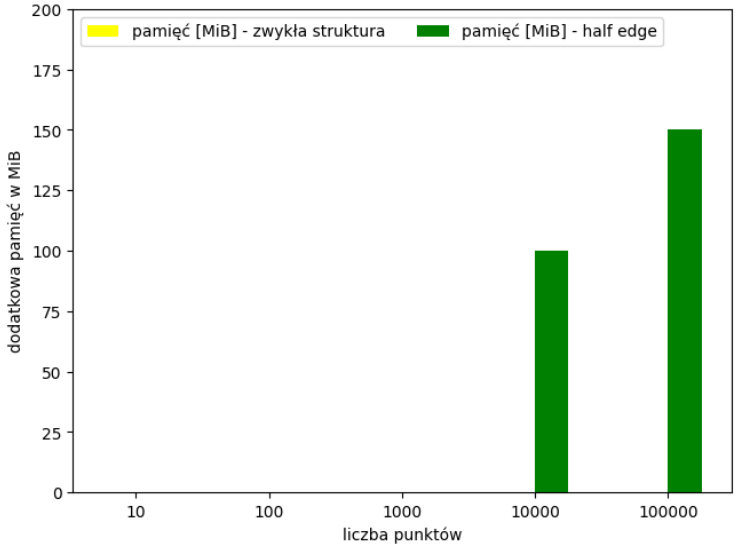


Diagram 15: pomiar pamięci używanej przez 3. operację, przeszukiwanie sąsiednich trójkątów w poszukiwaniu wierzchołka

	pamięć [MiB] - zwykła struktura	pamięć [MiB] - half edge	liczba wierzchołków
0	0.00000	0.00000	10
1	0.00000	0.00000	100
2	0.00000	0.00391	1000
3	0.00000	99.95312	10000
4	0.00000	150.03125	100000

Tabela 15: pomiar pamięci używanej przez 3. operację, przeszukiwanie sąsiednich trójkątów w poszukiwaniu wierzchołka

Każda operacja zachowała się bardziej stabilnie, to jest – nie pojawiły się większe artefakty w wynikach pomiarów. Zauważalny jest trend zużywania większej ilości pamięci dla coraz większych siatek, jedynie w przypadku struktury Half Edge. To jest akurat zrozumiałe – struktura wymaga alokowania pamięci dla obiektów odpowiadających krawędziom, oraz przypisanych im wskaźnikom. Ilość pamięci używanej przez każdą operację się zwiększyła – dla tej samej liczby wierzchołków, było więcej trójkątów, a tym samym krawędzi do zapisania. Dla każdej z operacji, przy 100 tysiącach wierzchołków, ilość zużywanej pamięci mieściła się w przedziale 130-200 MiB.

Podsumowanie

Dla wybranych do przetestowania operacji, wykorzystanie struktury Half Edge znacząco poprawia prędkość obliczeń. Wynika to z tego, że szukając sąsiednich wierzchołków/trójkątów, iteruje się tylko po wybranych, najbliższych krawędziach. Podczas gdy przy prostej strukturze danych, konieczne jest iterowanie po wszystkich trójkątach. Half Edge wymaga jednak dodatkowej pamięci, co przy wyjątkowo dużych siatkach może okazać się problemem. Mimo wszystko, zazwyczaj łatwiej o dużą ilość RAM-u w komputerze, niż o lepszy procesor, więc strata nie jest zbyt duża.

Wybór narzędzi do pomiaru pamięci miał spore znaczenie. Jedną z opcji, by uniknąć artefaktów, mogłoby być przeprowadzenie testów nie korzystając z notatnika Jupytera. Wtedy sposób, w jakim notatnik zarządza pamięcią, nie mógłby powodować żadnych problemów. Inną z opcji jest, by zamiast używać biblioteki memory-profiler, użyć funkcji getsizeof() z biblioteki sys, będącej w standardzie Pythona. To wymagałoby jednak dostosowania funkcji odpowiedzialnych za operacje, by za każdym razem mierzyć, ile pamięci zajmują zmienne wykorzystywane przez algorytm. Byłoby to nieco bardziej żmudne. Moduł Guppy3, służący do profilowania programów napisanych w Pythonie, też byłby opcją, ale trudniej było go wykorzystać w notatniku.

Struktura Half Edge jest szczególnie przydatna, gdy trzeba przeszukiwać więcej warstw sąsiednich trójkątów lub wierzchołków, co pokazały skrajnie duże różnice w wynikach pomiarów dla trzeciej operacji.

Generowanie siatek, generując wpierw wielokąty i licząc ich triangulacje, było skrajnie nieefektywnym podejściem, gdyż to była jedna z najpowolniejszych części programu służącego do testów, zaraz obok tej odpowiedzialnej za testowanie trzeciej operacji dla obu rodzajów siatek.

Bibliografia

- https://jerryyin.info/geometry-processing-algorithms/half-edge/
- https://cs184.eecs.berkeley.edu/sp19/article/15/the-half-edge-data-structure
- https://stackoverflow.com/questions/15365471/initializing-half-edge-data-structure-from-vertices
- https://kaba.hilvi.org/homepage/blog/halfedge/halfedge.htm
- Mark de Berg – „Computational Geometry – Algorithms and Applications”

Dodatek – analiza czasów tworzenia Half Edge

Przeprowadzono test, ile czasu zajmuje tworzenie struktury Half Edge. Tym razem jednak – z powodu awarii sprzętu – wykonano go na innej maszynie. Jej parametry podane są poniżej:

- Procesor: 11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz 3.00 GHz
- RAM: 8 GB
- System: Windows 11
- Wersja Python: 3.12.1
- GPU: zintegrowana z CPU

Zaprezentowano wynik testu dla drugiego typu siatki. Oba testy dają podobny wynik, ale dla tego typu siatki bardziej widoczne są zmiany wartości.

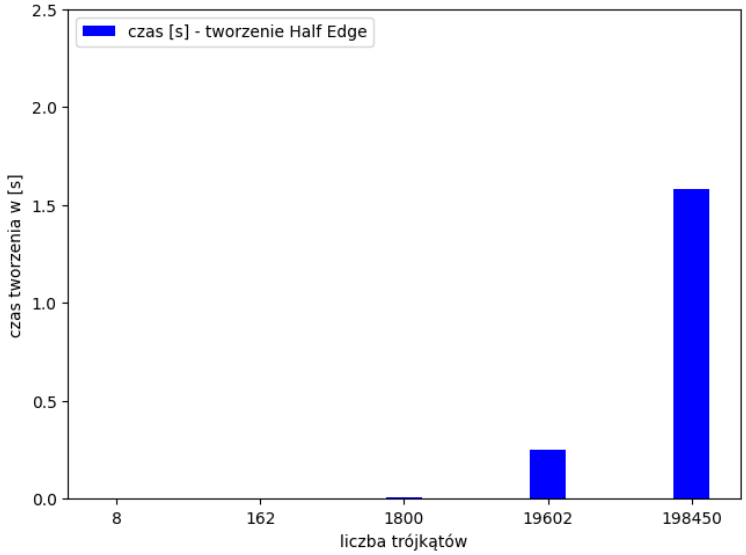


Diagram a: Czas tworzenia Half Edge, drugi typ siatki

	czas [s] - tworzenie Half Edge	liczba trójkątów
0	0.00007	8
1	0.00093	162
2	0.00715	1800
3	0.24757	19602
4	1.58166	198450

Tabela a: Czas tworzenia Half Edge, drugi typ siatki

Im większa liczba trójkątów, tym więcej czasu zajmuje konstrukcja struktury Half Edge. Należy o tym pamiętać, szczególnie gdy operacje na siatce chce się zrobić tylko raz. Dlatego, że czas konstrukcji wpływa na ogólną szybkość rozwiązania problemu. W przypadku, gdy chce się wykonywać operacje na jednej siatce wiele razy, jednorazowy koszt konstrukcji Half Edge nie zmienia aż tyle.