JIMP2 Sprawozdanie z projektu w języku C

Stanisław Dutkiewicz, Filip Kobus

Marzec 2024

Spis treści

1 Wstęp

Celem projektu jest stworzenie programu umożliwiającego znalezienie rozwiązania do wczytanego przez użytkownika labiryntu. Labirynt może być udostępniony zarówno w postaci pliku tekstowego jak i binarnego.

1.1 Problem Zadania

Głównym wyzwaniem projektu jest opracowanie efektywnego algorytmu, który umożliwi znalezienie w labiryncie ścieżki prowadzącej od punktu P do K. Maksymalny rozmiar labiryntu, jaki może być przekazany do rozwiązania, to 1024 × 1024 liczony po sciezkach, po ktorych mozna sie poruszac, co stawia przed nami zadanie efektywnego zarządzania pamięcią i optymalizacji algorytmu pod kątem czasu wykonania i zużycia zasobów.

Ważnym wymaganiem, dla części projektu realizowanej w języku C, jest ograniczenie dotyczące zużycia pamięci. Program w języku C nie powinien zużywać więcej niż 512 kB pamięci w trakcie całego swojego działania. To wymaganie podkreśla konieczność nie tylko implementacji skutecznego algorytmu przeszukiwania, ale również wydajnego zarządzania pamięcią oraz optymalnego projektowania struktur danych używanych w programie. Wyzwanie to wymaga głębokiego zrozumienia zarówno algorytmów, jak i niższopoziomowych aspektów zarządzania zasobami w programowaniu, co stanowi istotny element edukacyjny projektu.

Dodatkowo, program musi posiadać funkcjonalność wczytywania i zapisywania labiryntu w postaci pliku binarnego.

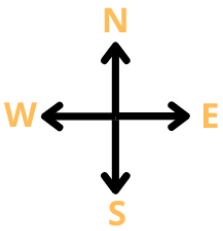
1.2 Sposób rozwiązania

Do rozwiązania tego problemu wykorzystane zostaną dwie metody przeszukiwania: BFS (Breadth-First Search) oraz DFS (Depth-First Search), z których użytkownik może wybrać jeden. Algorytmy te posłużą do eksploracji labiryntu w celu znalezienia rozwiązania.

DFS jest to algorytm, który eksploruje labirynt, poruszając się zgodnie z ustalonymi wcześniej priorytetami ruchów (omijając przy tym ściany), a gdy napotka ślepy zaułek, cofa się do ostatnio napotkanego węzła, aby spróbować innej ścieżki. Aby algorytm nie przekroczył zakładanego limitu pamięci, droga do każdego z tych zaułków będzie sukcesywnie usuwana z programu, co ograniczy wykorzystanie pamięci. Dodatkowo zostanie zaimplementowany w postaci iteracyjnej, mającej przewagę nad postacią rekurencyjną przez znacznie niższe zużycie zasobów. Poniżej przedstawiono przykładowe działanie algorytmu DFS.

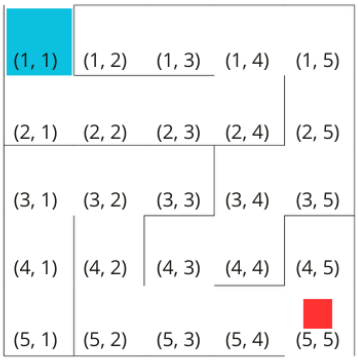
LEGENDA

- pole odwiedzone przez program i zaznaczone w pliku wejściowym
- pole obecnie analizowane
- (5, 5) - współrzędne pola
- wyjście z labiryntu
- 4. - numer iteracji
- 7. ... - po tej iteracji została pominięta przynajmniej jedna kolejna iteracja

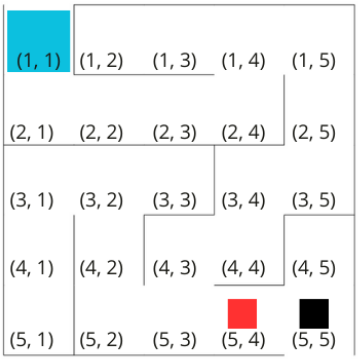


Priorytety ruchów:
W>N>E>S

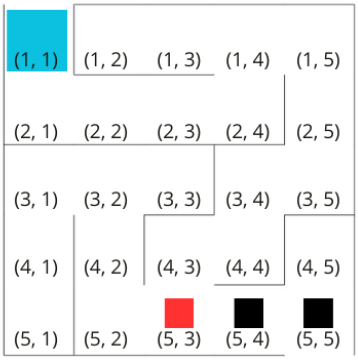
currCell - współrzędne obecnie analizowanej komórki
routesPossible - liczba możliwości ruchu w danej komórce
currPathLen/cPathLen - odległość między węzłami
nextMove - następny ruch jaki zostanie wykonany
Nodes = {(5,5)**}** - stos zawierający współrzędne napotkanych wcześniej węzłów
PathLens = {4**}** - stos zawierający odległości między węzłami
AllMoves = {3W, N**}** - stos zawierający instrukcje wyjściową (3 razy W i raz N)
popMovesMultiple(int x) - usuwa ze stosu instrukcji, x elementów
popPath(), popNodes() - zwraca i usuwa ze stosu ostatni element
pushNodes((5, 5)), pushPath(2) - wrzuca na stos element



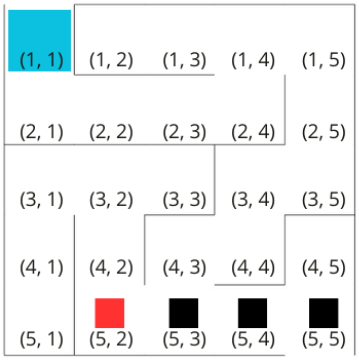
1.
currCell = (5,5)
routesPossible = 2
currPathLen = 1
nextMove = W
pushNodes(currCell)
pushPath(currPathLen)
Nodes = {(5,5)}
PathLens = {1}
AllMoves = {W}
cPathLen = 0



2.
currCell = (5,4)
routesPossible = 1
currPathLen = 1
nextMove = W
Nodes = {(5,5)}
PathLens = {1}
AllMoves = {2W}

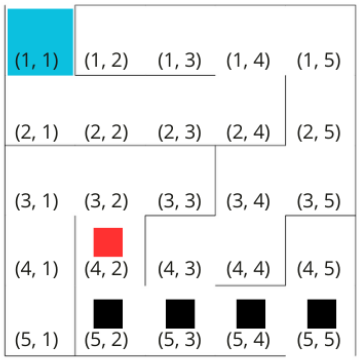


3.
currCell = (5,3)
routesPossible = 2
currPathLen = 2
nextMove = W
pushNodes(currCell)
pushPath(cPathLen)
Nodes = {(5,5), (5,3)}
PathLens = {1,2}
AllMoves = {3W}
cPathLen = 0

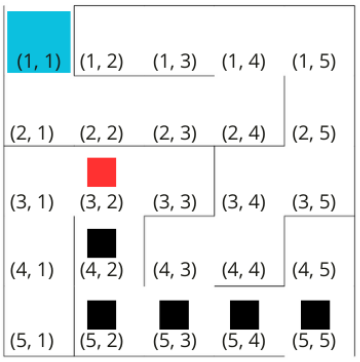


4.
currCell = (5,2)
routesPossible = 1
currPathLen = 1
nextMove = N
pushNodes(currCell)
pushPath(currPathLen)
Nodes = {(5,5),(5,3)}
PathLens = {1, 2}
AllMoves = {3W, N}

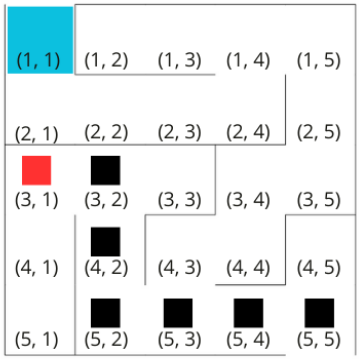
Rys.1 Skrótowa ilustracja działania algorytmu DFS



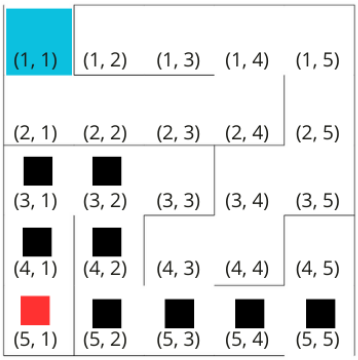
5.
currCell = (4,2)
routesPossible = 1
currPathLen = 2
nextMove = N
Nodes = {(5,5), (5,3)}
PathLens = {1, 2}
AllMoves = {3W, 2N}



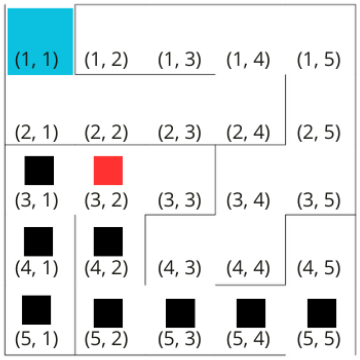
6.
currCell = (3, 2)
routesPossible = 2
currPathLen = 3
nextMove = W
pushNodes(currCell)
pushPath(currPathLen)
Nodes = {(5,5), (5,3), (3,2)}
PathLens = {1, 2, 3}
AllMoves = {3W, 2N, W}
currPathLen = 0



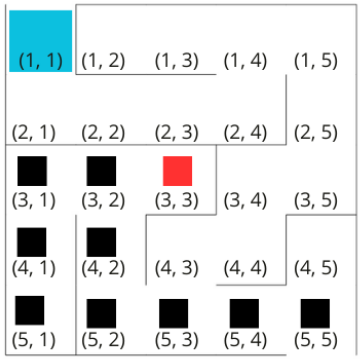
7. ...
currCell = (3,1)
routesPossible = 1
currPathLen = 1
nextMove = S
Nodes = {(5,5), (5,3), (3,2)}
PathLens = {1, 2, 3}
AllMoves = {3W, 2N, W, S}



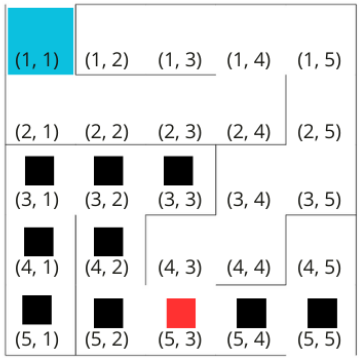
9.
currCell = (5,1)
routesPossible = 0
currPathLen = 3
nextMove = -
popMovesMultiple(cPathLen)
currentPathLen = popPath()
currCell = popNodes()
Nodes = {(5, 3)}
PathLens = {1, 2}
AllMoves = {3W, 2N}



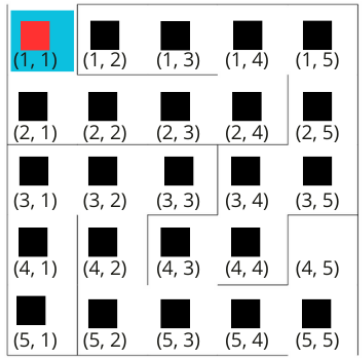
10.
currCell = (3,2)
routesPossible = 1
currentPathLen = 3
nextMove = E
Nodes = {(5,5),(5,3)}
PathLens = {1, 2}
AllMoves = {3W, 2N, E}



11.
currCell = (3,3)
routesPossible = 0
currentPathLen = 4
nextMove = -
popMovesMultiple(cPathLen)
cPathLen = popPath()
currCell = popNodes()
Nodes = {(5,5)}
PathLens = {1}
allMoves = {2W}



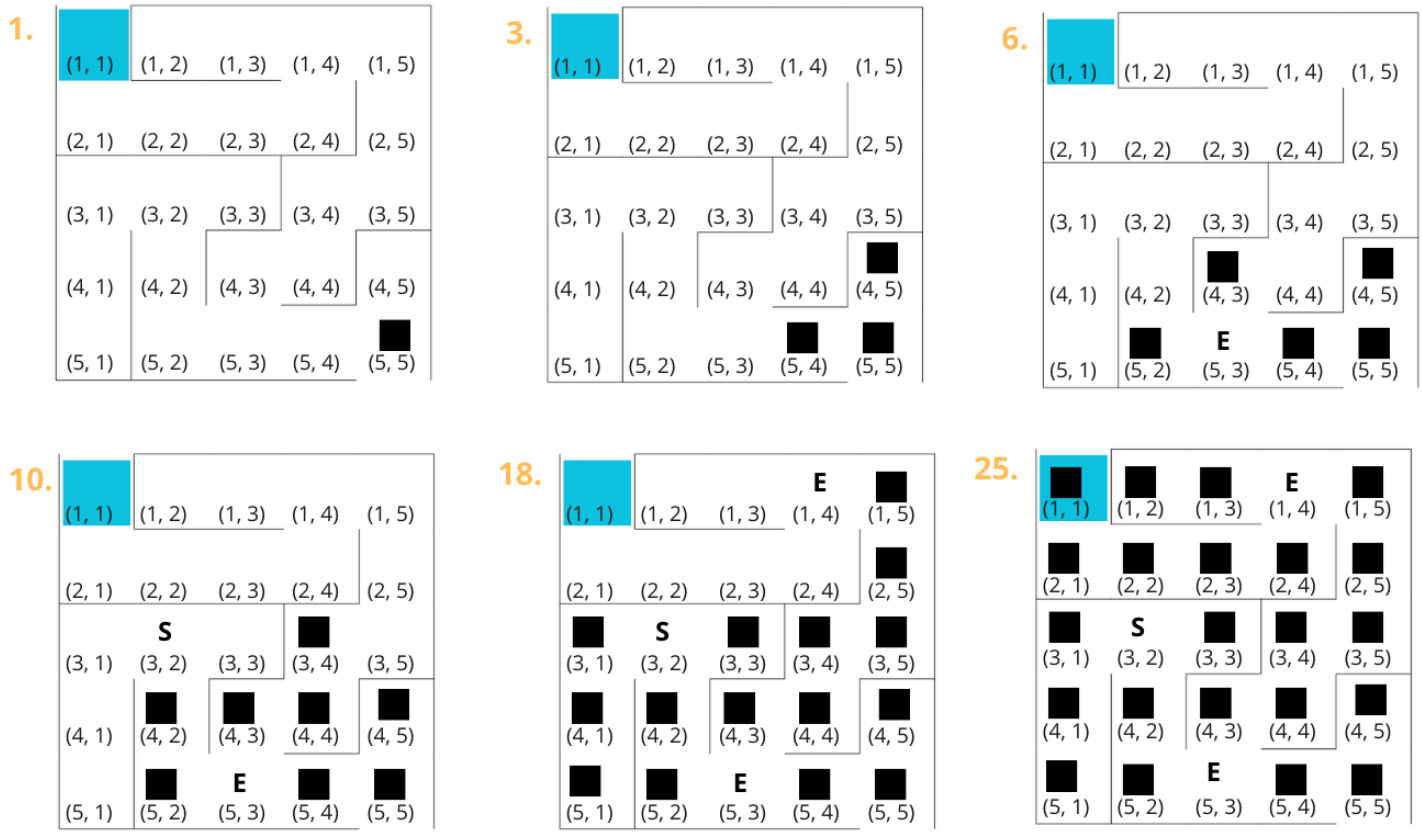
12. ...
currCell = (5,3)
routesPossible = 1
currentPathLen = 2
nextMove = N
Nodes = {(5,5)}
PathLens = {1}
AllMoves = {2W,N}



27.
currCell = (1,1)
routesPossible = 0
currentPathLen = 14
nextMove = -
allMoves = {2W, N, E,
N, E, 2N, W, S, 3W, N}
currCell == end

Rys.2 Skrótowa ilustracja działania algorytmu DFS cd.

Poniżej przedstawiono przykładowe działanie algorytmu DFS.



Rys.3 Skrótowa ilustracja działania algorytmu BFS

BFS algorytm polega na przeszukiwaniu labiryntu wszerz, zapisując na stworzonej wcześniej kolejce możliwe do przejścia komórki, a następnie sukcesywnie odwiedzając każdą z nich co tym samym pozwoli na znalezienie najkrótsze drogi do wyjścia. Jego przewagą na algorytmem DFS jest pewność znalezienia najkrótszej ścieżki, co w DFS zależy od ustawień początkowych oraz konfiguracji ścian labiryntu. Aby zaoszczędzić pamięć, kolejka jest tworzona w pliku tekstowym, przez co nie korzysta z zasobów pamięciowych programu.

Implementacja tych algorytmów w języku C, którą opracowaliśmy, wymagała szczególnej uwagi na efektywne zarządzanie pamięcią, zwłaszcza w kontekście ograniczenia do 512 kB pamięci dostępnej podczas działania programu. Aby sprostać temu wyzwaniu, kluczowe okazało się nie tylko zaprojektowanie samych algorytmów, ale również efektywne przechowywanie struktur danych, takich jak stosy, które służą do zapamiętywania odwiedzonych komórek i ścieżek w labiryncie.

Szczególnie istotnym elementem naszej strategii optymalizacji pamięci było zaimplementowanie stosu `char_stack`, który wykorzystuje utworzony podczas wywołaia programu plik tymczasowy który służy do zapisywania drogi pokonanej podczas przechodzenia labiryntu. Dzięki temu rozwiązaniu, zamiast obciążać pamięć operacyjną, stan algorytmu DFS jest przechowywany na dysku. To podejście znacząco zmniejsza zużycie pamięci języka C, pozwalając na dynamiczne zarządzanie zasobami i utrzymanie całkowitego zużycia pamięci znacznie poniżej ustalonego limitu 512 kB. Realizacja tej metody stanowi kluczowy czynnik umożliwiający efektywną pracę programu przy zachowaniu ograniczeń sprzętowych, nie wpływający przy tym na czas wykonywania programu.

1.3 Rodzaje plików wejściowych

Plik tekstowy: Zawiera symbole reprezentujące elementy labiryntu tj. wejście do labiryntu (P), wyjście z labiryntu (K), ściany (X) oraz miejsca, po których można się poruszać (spacja).

Plik binarny: Składa się z nagłówka pliku, informującego o: rozmiarach labiryntu, współrzędnych wejścia i wyjścia oraz binarnych reprezentacjach elementów labiryntu (takich samych jak te w wersji tekstowej). Następnie zamieszczono w niej sam labirynt w postaci przypominającej wersję tekstową z niewielką różnicą tzn. przed informacją o rodzaju komórki (ściana lub wolna przestrzeń), umieszczona została liczba jego wystąpień w danym miejscu co pozwoliło na znacznego ograniczenie rozmiarów pliku wejściowego. Ostatnim elementem pliku jest sekcja rozwiązania gdzie autor labiryntu może zamieścić rozwiązanie do labiryntu, choć nie jest to wymagane.

2 Implementacja

Implementacja projektu, realizowanego w języku C, skupiła się na rozwiązaniu problemu nawigacji przez labirynt z użyciem algorytmu przeszukiwania w głąb (DFS) oraz w szerz (BFS), przy jednoczesnym ograniczeniu zużycia pamięci. Kluczową decyzją projektową była modularna architektura systemu, która ułatwiła zarządzanie kodem oraz optymalizację pamięci. Szczególną uwagę poświęcono efektywnemu zarządzaniu danymi, w tym zastosowaniu stosu znaków, który zapisuje postęp w pliku tymczasowym, minimalizując w ten sposób konsumpcję pamięci programu. Poniżej są przedstawione kluczowe moduły projektu, które razem tworzą spójny system zdolny do skutecznego rozwiązywania labiryntów.

2.1 Moduły

Projekt składa się z kilku kluczowych modułów, które współpracują ze sobą w celu rozwiązania problemu nawigacji przez labirynt:

- **Wczytywanie labiryntu (`load_maze.h`, `load_maze.c`)**: Ten moduł jest odpowiedzialny za inicjalizację labiryntu poprzez odczyt z pliku tekstowego. Funkcje zawarte w tym module pozwalają na określenie wymiarów labiryntu oraz identyfikację lokalizacji punktów startowego i końcowego. Te dane są zapisywanie w specjalnej strukturze, która jest używana bardzo często. Jest to krytyczny pierwszy krok w procesie rozwiązywania labiryntu, ponieważ dostarcza struktury danych niezbędnej do nawigacji.
- **Wczytywanie labiryntu z pliku binarnego (`binary.h`, `binary.c`)**: Moduł ten jest wykorzystywany w przypadku kiedy dane labiryntu zostaną wprowadzone w postaci pliku binarnego. Funkcje które implementuje to:
  - Odczytywanie pliku binarnego
  - Przetwarzanie labiryntu z formy binarnej na tekstową
  - Wypisanie labiryntu wraz z rozwiązaniem w formie pliku binarnego
- **Zarządzanie argumentami (`manage.h`, `manage.c`)**: Ten moduł odpowiada za zarządzanie argumentami wprowadzanymi przez użytkownika - korzystając z funkcji getopt przyporządkowuje wprowadzonem akronimowi funkcjonalność algorytmu.
- **Algorytmy (`dfs.h`, `dfs.c` oraz `bfs.h`, `bfs.c`)**: Zawierają serce logiki nawigacyjnej programu, wykorzystując poniżej wspomniane moduły. Pierwszy z nich - DFS ustala priorytety kierunków ruchu na podstawie aktualnej pozycji i kierunku docelowego. Dodatkowo zawiera funkcję move, która dla podanej komórki, sprawdza jakie są dostępne ruchy, zwraca ich ilość oraz wybiera z pośród dostępnych możliwości, kolejny ruch, wraz ze współrzędnymi następnej komórki. Drugi algorytm - BFS, również składa się z funkcji moveb, która analizuje sąsiednie komórki danej komórki i zapisuje w kolejce wszystkie dostępne możliwości. W tym

algorytmie kierunki ruchów są ustawione domyślnie, ponieważ nie ma to wpływu na działanie algorytmu i znajdowanie ścieżki. Aby odczytać finalne rozwiązanie, do każdego z węzłów wpisywany jest kierunek przeciwny do kierunku wejścia np. jeżeli węzeł został odwiedzony od północy do wpisujemy do niego S (South - z ang. południe).

- **Stos znaków (`char_stack.h`, `char_stack.c`)**: Moduł ze stosem znaków odgrywa kluczową rolę w śledzeniu i zarządzaniu sekwencjami ruchów w labiryncie. W odróżnieniu od tradycyjnych implementacji, stos ten wykorzystuje plik tymczasowy na dysku do zapisywania postępu, co pozwala na znaczne ograniczenie zużycia pamięci operacyjnej. Każdy ruch w labiryncie, reprezentowany przez charakterystyczny znak ('N', 'S', 'E', 'W'), jest zapisywany bezpośrednio w tym pliku, dzięki czemu można dynamicznie zarządzać historią eksploracji bez obawy o przekroczenie dostępnego limitu pamięci 512 kB. Jest on wykorzystywany w algorytmie DFS.

- **Stos liczb całkowitych (`int_stack.h`, `int_stack.c`)**: Ten moduł implementuje stos, dla liczb całkowitych (typ 'int'). Jest on używany do przechowywania współrzędnych napotkanych podczas przechodzenia labiryntu węzłów, które jak wynika z wymagań zadania, nie mogą być większe niż 4mln., co mieści się w zakresie zmiennej typu int. Stos operuje na tablicy dynamicznej, która podczas zbliżania się do swojego maksymalnego rozmiaru zwiększa swoją pojemność dwukrotnie. Jest on wykorzystywany w algorytmie DFS.

- **Kolejka liczb całkowitych(`queue.h`, `queue.c`)**: Ten moduł implementuje kolejkę dla liczb całkowitych (typ 'int'), która wykorzystuje plik tekstowy do przechowywania danych. Kolejka jest używana do przechowywania współrzędnych napotkanych podczas przechodzenia labiryntu węzłów, zgodnie z wymaganiami zadania. W przypadku, gdy liczba współrzędnych przekracza rozmiar pamięci RAM, wykorzystanie pliku tekstowego pozwala na elastyczne zarządzanie danymi. Kolejka umożliwia operacje dodawania elementów na koniec kolejki oraz usuwania elementu z początku kolejki.

2.2 Funkcjonalność

Program oferuje następujące funkcjonalności:

- **Dynamiczne wczytywanie labiryntów**: Możliwość wczytania dowolnego labiryntu zarówno z pliku tekstowego jak i binarnego, z automatycznym rozpoznaniem rozmiaru oraz lokalizacji startu i mety.

- **Wybór algorytmu**: Możliwość wybrania algorytmu rozwiązującego labirynt (DFS lub BFS).

- **Efektywne znajdowanie ścieżki**: Użycie algorytmu przeszukiwania w głąb (DFS), aby znaleźć drogę przez labirynt, z minimalnym zużyciem pamięci.

- **Efektowne znajdowanie ścieżki**: Użycie algorytmu przeszukiwania w szerz (BFS), aby znaleźć najkrótszą drogę przez labirynt.
- **Ręczne ustalanie priorytetu**: W algorytmie DFS istniej możliwość ustalenia priorytetu współrzędnej horyzontalnej lub wertykalnej który będzie istotny podczas tworzenia kolejności wyboru ruchów. Jest to dostępne na samej górze pliku `dfs.h`
- **Eksport wyników**: Możliwość zapisania wykonanych przez algorytm kroków do pliku (kroki.txt), co ułatwia analizę i prezentację rozwiązania. Również zapisywany jest plik binarnego z labiryntem oraz jego rozwiązaniem.
- **Widowiskowy sposób zapisu wyniku**: Wynik działania algorytmu, oprócz postaci kolejno zapisanych instrukcji, jest zapisywany również jako plik tekstowy zawierający labirynt wejściowy wraz z naniesioną na niego ścieżką, co umożliwia szybkie sprawdzenie rezultatu wywołania programu.

2.3 Uruchamianie programu

Aby uruchomić program, użytkownik musi umieścić w folderze `source`, plik zawierający labirnyt (w rozszerzeniu .txt lub .bin) lub skorzystać z domyślnego pliku maze.txt umieszczonego w katalogu głównym programu. Struktura labiryntu w pliku tekstowym, powinna wykorzystywać następujące oznaczenia:

- **P** - punkt wejścia do labiryntu,
- **K** - punkt wyjścia z labiryntu,
- **X** - ściana,
- **spacja** - wolne miejsce, po którym można się poruszać.

Program można uruchomić wykonując następujące kroki:

1. Otwórz terminal w systemie Linux.
2. Przejdź do katalogu zawierającego pliki programu za pomocą polecenia `cd ścieżka/do/katalogu`.
3. Umieść plik z labiryntem w katalogu `source`.
4. Skompiluj program używając kompilatora make: `make all`.
5. Uruchom program poleceniem: `./app`.
6. Wczytaj swój labirynt: `./app -n "plik.txt"` lub `./app -n "plik.bin"`.
7. Wybierz algorytm: `./app -a dfs` lub `./app -a bfs`.
8. Uzyskaj pomoc: `./app -h`.

Uwaga: Dla poprawnego działania programu, wprowadzony plik wejściowy musi być obecny w katalogu `source` i zawierać prawidłowo sformatowany labirynt. W przypadku braku pliku lub błędów w jego formacie, program zakończy działanie z odpowiednim komunikatem błędu. Program można wywołać również bez argumentów, wtedy zostanie wczytana domyślna plansza maze.txt, a labirynt rozwiąże algorytm dfs.
Upewnij się, że środowisko uruchomieniowe spełnia wszystkie wymagania niezbędne do kompilacji i wykonania programu, w tym posiadanie zainstalowanego kompilatora cc i standardowych bibliotek języka C.

2.4 Sposób działania algorytmów

Program rozpoczyna od wybrania algorytmu (BFS lub DFS), wczytania labiryntu za pomocą modułu `load_maze`, a następnie inicjalizacji struktury danych do przechowywania informacji o ruchach. Kolejnym krokiem jest ustalenie priotytetów ruchów na podstawie współrzędnych wejścia oraz wyjścia z labiryntu. Następnie dany algorytm, przy pomocy funkcji firstMove podejmuje pierwszy krok, tak aby nie wyjść poza sciany labiryntu, a w dalszej kolejności tą rolę przejmuje funkcja move, wykonywana w pętli `while`, tak długo, aż obecna komórka będzie różna od komórki mety. Podczas eksploracji, każdy wykonany ruch jest oznaczany symbolem: '-', jako miejsce do których algorytm nie powróci. Dodatkowo algorytm BFS oznacza każdy z odwiedzonych węzłów kierunkiem wejścia do tego węzła. Po znalezieniu ścieżki, wykonane ruchy są eksportowane do pliku (output/kroki.txt), tworzony jest plik labiryntZeSciezka w folderze output, który zawiera labirynt z zaznaczoną symbolem '*' drogą do wyjścia, a program zamyka otwarte zasoby, czyści planszę wejściową z zaznaczonych ruchów i kończy działanie.
Dzięki modularnej strukturze, projekt ten jest łatwo rozszerzalny i umożliwia dodawanie nowych funkcji, takich jak graficzny interfejs użytkownika czy zaawansowane algorytmy przeszukiwania.

3 Testy

Podczas oceny wydajności naszego rozwiązania skupiliśmy się na testowaniu czterech labiryntów o różnych rozmiarach: 10x10, 100x100, 512x512 oraz 1024x1024. Dla każdego z tych labiryntów przeprowadziliśmy testy mające na celu zmierzenie zużycia pamięci oraz czasu potrzebnego na znalezienie rozwiązania. Testy przeprowadzone zostały w środowisu Linux Ubuntu, wersji 18.03, stosując odpowiednio moduł time do mierzenia czasu oraz program Valgrind pozwalający na zmierzenie maksymalnego zużycia pamięciowego. Poniżej przedstawiamy wyniki tych testów. Testy dotyczą jedynie algorytmu DFS.

3.1 Labirynt 10x10

Zużycie pamięci: 14,859 B
Czas trwania: ok. 0.009 s

3.2 Labirynt 100x100

Zużycie pamięci:16,839 B
Czas trwania: ok. 0.06 s

3.3 Labirynt 512x512

Zużycie pamięci: 31,103 B
Czas trwania: ok. 4.1 s

3.4 Labirynt 1024x1024

Zużycie pamięci:47,488 B
Czas trwania: ok. 7.1-15.4 s

Wnioski z tych testów będą miały kluczowe znaczenie dla dalszego rozwoju naszego projektu. Pozwalają one na zrozumienie, jak skala labiryntu wpływa na wydajność algorytmu i efektywność zarządzania pamięcią. Wyniki testów mogą również wskazywać potencjalne obszary do optymalizacji, aby jeszcze bardziej poprawić wydajność naszego rozwiązania w przyszłości.

4 Wnioski

Realizacja tego projektu na początku semestru stanowiła dla nas doskonałą okazję do odświeżenia i pogłębienia umiejętności programowania w języku C.

Jednym z największych wyzwań, przed którymi stanęliśmy, było efektywne zarządzanie pamięcią. W miarę rozwoju projektu zdecydowaliśmy się na znaczącą optymalizację struktury danych, przede wszystkim poprzez przejście z listy połączonej na dynamiczne tablice. Spowodowało to znaczący spadek zajmowanej pamięci, mieszącej się w zaplanowanym limicie, jednak dopiero transformacja modułu `char_stack.c` okazała się satysfakcjonująca - zamiast stosować tradycyjne struktury stosu, opracowaliśmy mechanizm wykorzystujący plik tymczasowy do zapisywania sekwencji ruchów. Dzięki temu znacznie zredukowaliśmy zużycie pamięci operacyjnej, co pozwoliło nam na znaczące obniżenie używanych zasobów i tym samym, spełnienie wymagań projektowych dotyczących ograniczeń pamięciowych.

Najbardziej wymagającym zadaniem, przed którym stanęliśmy, było zapisanie labiryntu wraz z wynikiem w pliku binarnym. Choć zastosowaliśmy najlepsze znane nam techniki i algorytmy, ciągle pozostaje pewna niepewność co do optymalności i poprawności tej części projektu. To doświadczenie uświadomiło nam, że nawet po skutecznej implementacji innych modułów, całościowe zrozumienie i prawidłowa realizacja zapisu danych w formacie binarnym może stanowić wyzwanie i wymagać dalszego doskonalenia naszych umiejętności programistycznych.