

Politechnika Świętokrzyska
Wydział Elektrotechniki, Automatyki i Informatyki

Dokumentacja projektu zespołowego
Wizualizacja wyszukiwania drogi w labiryncie

Filip Stępień Rafał Grot

Nr indeksu: 094117 Nr indeksu: 094046

Informatyka, grupa 3ID11B

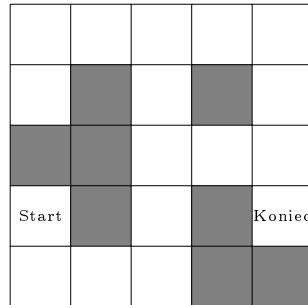
20 czerwca 2025

Spis treści

| | | |
|----------|---|-----------|
| 1 | Wstęp | 3 |
| 2 | Generowanie labiryntu | 4 |
| 2.1 | Algorytm Prima | 5 |
| 2.2 | Algorytm Kruskala | 7 |
| 2.3 | Algorytm <i>przeszukiwania w głąb</i> | 10 |
| 3 | Wyszukiwanie drogi | 12 |
| 3.1 | Stany węzłów | 12 |
| 3.2 | Algorytm DFS | 13 |
| 3.2.1 | Opis działania algorytmu | 13 |
| 3.2.2 | Inicjalizacja | 13 |
| 3.2.3 | Funkcja rekurencyjna DFS | 13 |
| 3.2.4 | Proces budowania ścieżki | 13 |
| 3.2.5 | Możliwe stany węzłów | 14 |
| 3.2.6 | Złożoność obliczeniowa | 14 |
| 3.3 | Algorytm BFS | 17 |
| 3.3.1 | Opis działania algorytmu | 17 |
| 3.3.2 | Inicjalizacja | 17 |
| 3.3.3 | Proces przeszukiwania BFS | 17 |
| 3.3.4 | Proces budowania ścieżki | 17 |
| 3.3.5 | Złożoność obliczeniowa | 18 |
| 3.3.6 | Właściwości algorytmu | 18 |
| 3.4 | Algorytm A* | 23 |
| 3.4.1 | Opis działania algorytmu | 23 |
| 3.4.2 | Inicjalizacja | 23 |
| 3.4.3 | Funkcja heurystyczna | 23 |
| 3.4.4 | Główna pętla algorytmu | 23 |
| 3.4.5 | Budowanie ścieżki | 24 |
| 3.4.6 | Złożoność obliczeniowa | 24 |
| 4 | Instrukcja obsługi aplikacji | 28 |
| 4.1 | Uruchamianie | 28 |
| 4.2 | Interfejs użytkownika | 28 |
| 4.3 | Przeprowadzanie symulacji | 30 |
| 5 | Porównanie algorytmów wyszukiwania ścieżki | 31 |
| 5.1 | Symulacja | 31 |
| 5.2 | Interpretacja wyników | 33 |
| 6 | Zakończenie i wnioski końcowe | 34 |

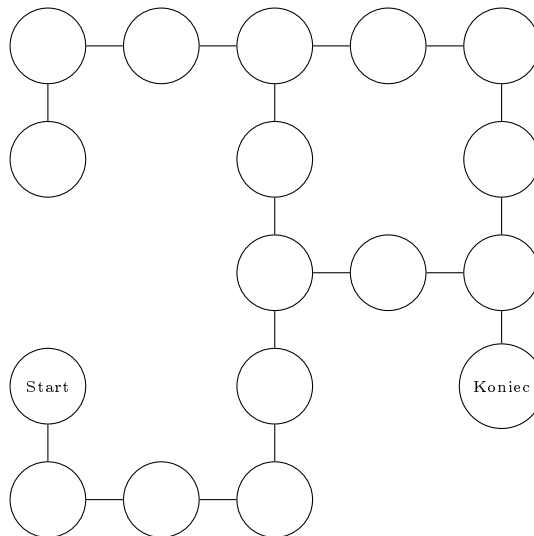
1 Wstęp

Celem projektu jest stworzenie aplikacji umożliwiającej generowanie dwuwymiarowego labiryntu oraz wizualizację procesu wyszukiwania ścieżki pomiędzy dwoma punktami. Labirynt w kontekście projektu to struktura siatki, gdzie każde pole może stanowić przejście lub ścianę. Przykładowy labirynt pokazano na rysunku 1.



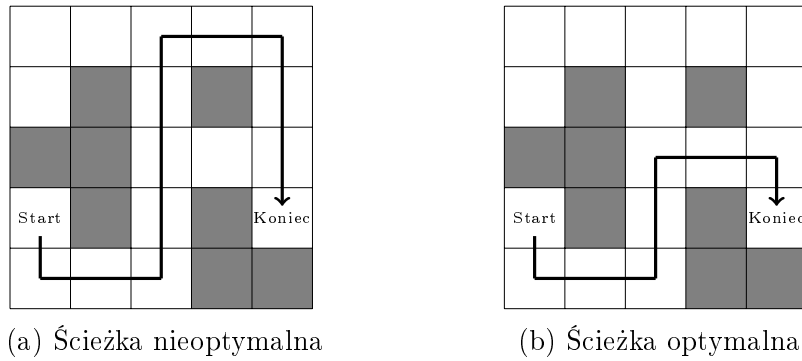
Rysunek 1: Przykładowy labirynt 5x5 z zaznaczonym startem i końcem, gdzie białe pole - przejście, czarne - ściana.

Można zauważyć, że taka struktura jest reprezentacją grafu, gdzie pola odpowiadają wierzchołkom, a krawędzie łączą sąsiadujące pola przejściowe. Reprezentacja labiryntu w formie grafu została przedstawiona na rysunku 2.



Rysunek 2: Graf reprezentujący labirynt z Rysunku 1.

W projekcie istotne jest porównanie różnych algorytmów wyszukiwania ścieżki, które pozwalają znaleźć trasę między dwoma punktami. W najlepszym przypadku celem jest znalezienie ścieżki *optymalnej*, czyli takiej, która minimalizuje liczbę kroków, co w kontekście grafu o jednakowych wagach krawędzi sprowadza się do znalezienia drogi o minimalnej długości. Ścieżkę *optymalną* oraz *nieoptymalną* ukazuje rysunek 3.



Rysunek 3: Porównanie ścieżek w labiryncie.

2 Generowanie labiryntu

Do generowania labiryntów kluczowe jest zrozumienie idei drzew rozpinających.

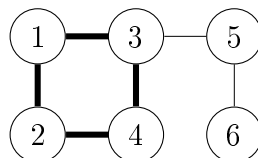
Drzewo rozpinające [2] to taki podgraf oryginalnego grafu, który zawiera wszystkie jego wierzchołki, a jednocześnie jest spójny i nie zawiera cykli.

Spójność grafu [1] oznacza, że między każdą parą wierzchołków istnieje co najmniej jedna ścieżka, czyli można przejść z dowolnego wierzchołka do dowolnego innego, poruszając się po krawędziach grafu. Różnicę między grafem spójnym oraz niespójnym przedstawia rysunek 4.



Rysunek 4: Przykłady grafów spójnych i niespójnych.

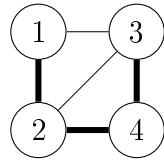
Cyklem [1] nazywamy ścieżkę zaczynającą się i kończącą w tym samym wierzchołku, w której żadna krawędź ani wierzchołek (poza początkiem i końcem) się nie powtarza. Obecność cykli oznacza, że można okrążyć pewien obszar grafu i wrócić do punktu startu inną drogą, co w kontekście labiryntu oznacza istnienie pętli. W drzewie rozpinającym takich pętli nie ma, co gwarantuje unikalną ścieżkę między dowolnymi dwoma wierzchołkami. Przykładowy graf zawierający cykl został pokazany na rysunku 5.



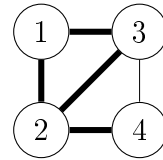
Rysunek 5: Graf zawierający cykl (pogrubiona linia).

Minimalne drzewo rozpinające [2] (ang. *MST*, *minimum spanning tree*) to takie drzewo rozpinające, dla którego suma wag krawędzi jest najmniejsza spośród wszystkich możliwych drzew rozpinających danego grafu. W grafach nieważonych, jak te stosowane

przy modelowaniu labiryntów, wszystkie krawędzie są traktowane jako równe, dlatego każde drzewo rozpinające o minimalnej liczbie krawędzi spełnia warunki *MST*. Przykładowe minimalne drzewo rozpinające dla grafu z równoważnymi krawędziami pokazuje rysunek 6.



(a) Ścieżka będąca minimalnym drzewem rozpinającym (3 kroki).



(b) Ścieżka nie stanowiąca minimalnego drzewa rozpinającego (4 kroki).

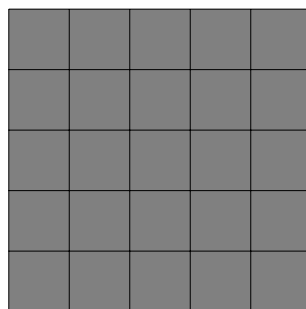
Rysunek 6: Ścieżki spełniające oraz nie spełniające założeń *MST* (pogrubione linie).

Algorytmy generujące labirynty sprowadzają się właśnie do wyznaczenia wspomnianego drzewa rozpinającego. Uzyskane drzewo określa, które krawędzie (ścieżki) są dostępne, a które stanowią ściany labiryntu.

2.1 Algorytm Prima

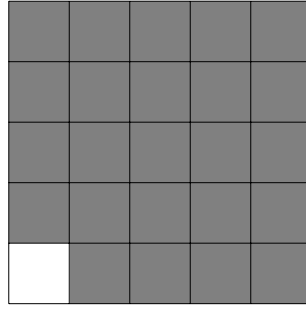
Algorytm Prima [2] w kontekście generowania labiryntu działa poprzez stopniowe budowanie połączeń między komórkami planszy. W każdej iteracji wybierana jest losowa krawędź prowadząca z odwiedzonej komórki do jednej z sąsiednich, jeszcze nieodwiedzonych. Wybrana komórka zostaje następnie połączona z dotychczasowym obszarem labiryntu. Proces ten powtarzany jest aż do momentu, gdy wszystkie komórki zostaną połączone, tworząc spójną strukturę bez cykli. Szczegółowy przebieg algorytmu wygląda następująco:

1. Na początku tworzona jest plansza, w której wszystkie komórki są oznaczone jako ściany. Rysunek 7 przedstawia początkowy układ planszy w całości wypełnionej ścianami.



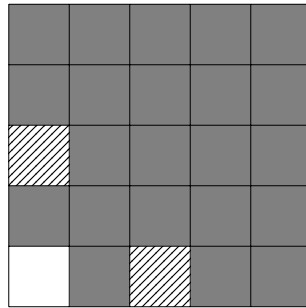
Rysunek 7: Cała plansza stanowi ściany.

2. Następnie wybierane jest losowe pole, które zostaje oznaczone jako przejście. Rysunek 8 przedstawia wybrane losowo pole startowe.



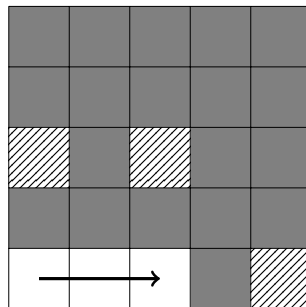
Rysunek 8: Losowe pole startowe.

3. Do zbioru krawędzi dodawane są sąsiednie komórki, do których można przejść bezpośrednio z pola startowego. Za sąsiednie uznaje się komórki oddalone o jedno pole w pionie lub poziomie. Ilustrację tego etapu przedstawiono na rysunku 9.



Rysunek 9: Wybór sąsiednich pól.

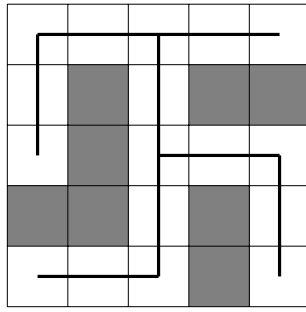
4. Losowana jest jedna krawędź ze zbioru potencjalnych przejść. Jeśli prowadzi ona do nieodwiedzanego pola, tworzy się przejście między bieżącym polem a nowym (usuwana jest ściana między nimi), a nowe pole zostaje oznaczone jako przejście. Następnie do zbioru krawędzi dodawani są sąsiedzi nowo odwiedzonego pola. Ilustrację tego etapu przedstawiono na rysunku 10.



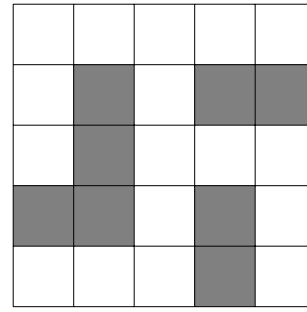
Rysunek 10: Utworzenie krawędzi do sąsiedniego pola.

5. Proces powtarza się, aż zbiór krawędzi stanie się pusty, co oznacza, że wszystkie komórki zostały połączone.

Na rysunku 11a przedstawiono wyznaczanie kolejnych krawędzi labiryntu, a na rysunku 11b końcowy labirynt powstały na podstawie tych krawędzi.



(a) Wyznaczanie kolejnych krawędzi labiryntu.



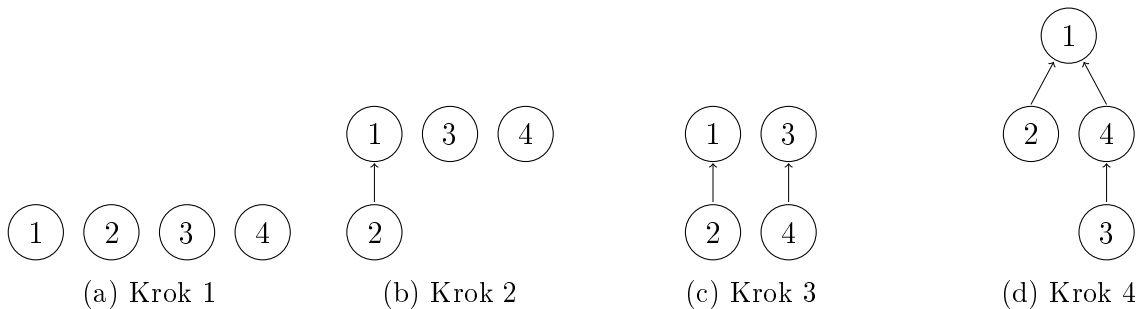
(b) Labirynt powstały na podstawie wyznaczonych krawędzi.

Rysunek 11: Kolejne kroki działania algorytmu.

2.2 Algorytm Kruskala

Algorytm Kruskala [2] do znalezienia minimalnego drzewa rozpinającego wykorzystuje strukturę zbiorów rozłącznych (ang. *Disjoint Set*).

Struktura *Disjoint Set* reprezentuje rozłączne zbiory elementów za pomocą drzew. Na początku każdy element tworzy pojedynczy, jednoelementowy zbiór, którego reprezentantem jest korzeń drzewa. Główna operacja na tej strukturze, *unia*, łączy dwa zbiory, tworząc jedno drzewo, w którym korzeń jednego zbioru staje się potomkiem korzenia drugiego. W ten sposób zbiory są scalane. Schematyczne przedstawienie scalania zbiorów znajduje się na rysunku 12.



Rysunek 12: Kolejne kroki scalania zbioru

Projekt wykorzystuje zmodyfikowaną wersję algorytmu Kruskala. W klasycznym wariacie algorytm losowo wybiera krawędź, czyli parę sąsiadujących komórek, które mogą zostać połączone ścieżką. Jeśli komórki te należą do różnych zbiorów, są one łączone w jeden zbiór, a między nimi tworzone jest przejście.

W prezentowanej wersji algorytmu zamiast losowo wybierać krawędzie, iteruje się w losowej kolejności po wszystkich komórkach planszy. Dla każdej komórki rozpatrywani są jej sąsiedzi – jeśli należą do innych zbiorów, bieżąca komórka zostaje przekształcona w ścieżkę, a sąsiadujące zbiory zostają połączone.

Kluczową różnicą w porównaniu do klasycznego podejścia jest sposób zakończenia algorytmu. W tradycyjnej wersji wszystkie komórki zostają połączone w jeden wspólny zbiór. W tym przypadku natomiast, ze względu na lokalny charakter działania, istnieje wysokie prawdopodobieństwo, że w wyniku działania algorytmu pozostanie wiele niepołączonych zbiorów. Mimo to, końcowy układ labiryntu wizualnie przypomina ten uzyskany metodą klasyczną.

Poszczególne kroki generowania labiryntu przebiegają następująco:

1. Tworzona jest siatka, w której wszystkie pola są początkowo oznaczone jako przejścia. Warto zaznaczyć, że nie ma znaczenia, czy algorytm rozpoczyna z planszą wypełnioną przejściami, a następnie tworzy ściany, czy odwrotnie — z planszą wypełnioną ścianami, w której tworzone są przejścia. Efekt końcowy pozostaje taki sam. Przykład planszy z samymi przejściami przedstawiono na rysunku 13.

| | | | | |
|--|--|--|--|--|
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

Rysunek 13: Cała plansza stanowi przejścia.

2. Każde pole planszy jest osobnym zbiorem w strukturze zbiorów rozłącznych. Na tym etapie żadna komórka nie jest jeszcze połączona z inną, co zostało przedstawione na rysunku 14.

| | | | | |
|----------|----------|----------|----------|----------|
| A | F | K | P | U |
| B | G | L | Q | V |
| C | H | M | R | W |
| D | I | N | S | X |
| E | J | O | T | Y |

Rysunek 14: Każdy zbiór jest oznaczony unikalną literą.

3. Komórki planszy są rozpatrywane w losowej kolejności:
 - (a) Dla wybranej komórki określa się bezpośrednich sąsiadów. W tym przypadku są to pola bezpośrednio przyległe w górę, doł, lewo lub prawo - inaczej niż przyjęto w algorytmie Prima. Przykład takiej sytuacji przedstawiono na rysunku 15.

| | | | | |
|----------|----------|----------|----------|----------|
| A | F | K | P | U |
| B | G | L | Q | V |
| C | H | M | R | W |
| D | I | N | S | X |
| E | J | O | T | Y |

Rysunek 15: Losowa komórka (oznaczona kółkiem) i jej sąsiedzi (zakreskowani).

- (b) Jeśli wszyscy sąsiedzi należą do różnych zbiorów to wylosowana komórka staje się ścianą, a jej sąsiedzi są łączeni w jeden zbiór. Schemat łączenia komórek w zbiory pokazano na rysunku 16.

| | | | | |
|---|---|---|---|---|
| A | F | K | P | U |
| B | G | L | Q | V |
| C | H | M | R | W |
| H | I | H | S | X |
| E | H | O | T | Y |

Rysunek 16: Łączenie komórek w zbiory. Reprezentantem zbioru jest pierwszy dodany element (tutaj komórka H), choć może to być dowolna komórka z grupy.

- (c) Jeśli chociaż dwaj sąsiedzi należą do tego samego zbioru, komórka nie zostaje oznaczona jako ściana – pozostaje przejściem, co przedstawiono na rysunku 17.

| | | | | |
|---|---|---|---|---|
| A | F | K | P | U |
| B | G | L | Q | V |
| C | H | M | R | W |
| H | I | H | S | X |
| E | H | O | T | Y |

(a) Wylosowanie kolejnej komórki.

| | | | | |
|---|---|---|---|---|
| A | F | K | P | U |
| B | G | L | Q | V |
| C | H | M | R | W |
| H | I | H | S | X |
| E | H | O | T | Y |

(b) Pozostawienie przejścia.

Rysunek 17: Wylosowana komórka ma już dwóch takich samych sąsiadów — scalanie nie następuje. Przejście poglądowo oznaczono jasnoszarym kolorem, aby zaznaczyć, że zostało odwiedzone.

4. Proces powtarza się, aż każda komórka zostanie przetworzona. Przykładowe dalsze kroki algorytmu przedstawia rysunek 18, a rysunek 19 ukazuje potencjalny schemat wygenerowanego labiryntu.

| | | | | |
|---|---|---|---|---|
| A | F | K | P | U |
| B | G | L | Q | V |
| C | H | M | R | W |
| H | I | H | S | X |
| E | H | O | T | Y |

(a) Kolejna komórka bez unikalnych sąsiadów.

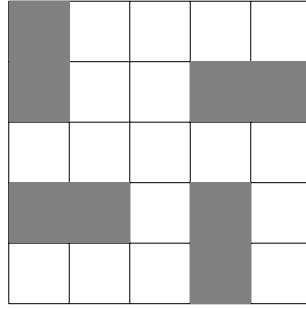
| | | | | |
|---|---|---|---|---|
| A | F | K | P | U |
| B | G | L | Q | V |
| C | H | M | R | W |
| H | I | H | S | X |
| C | H | O | T | Y |

(b) Komórka należąca do scalonego zbioru.

| | | | | |
|---|---|---|---|---|
| A | F | K | P | U |
| B | G | P | Q | P |
| C | H | M | P | W |
| H | C | H | S | X |
| C | H | O | T | Y |

(c) Scalanie kolejnego osobnego zbioru.

Rysunek 18: Przykładowe kolejne iteracje algorytmu. Utworzone zbiory oznaczono unikalnymi liniami.

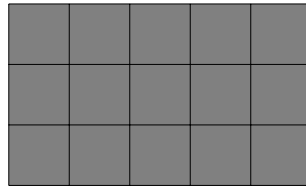


Rysunek 19: Przykładowy wygląd końcowego, wygenerowanego labiryntu.

2.3 Algorytm przeszukiwania w głąb

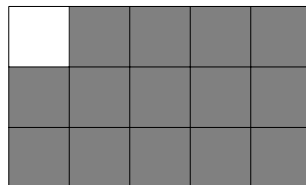
Algorytm przeszukiwania w głąb (ang. *Depth-First Search*, DFS) [2] to klasyczna metoda eksploracji grafu, która w kontekście generowania labiryntu pozwala na systematyczne zagłębianie się w kolejne, jeszcze nieodwiedzone komórki planszy z wykorzystaniem mechanizmu rekurencji. Algorytm rozpoczyna od wybranej komórki startowej i rekursywnie odwiedza sąsiadujące, nieodwiedzone komórki, usuwając ściany między nimi w celu utworzenia ścieżek. Algorytm działa w następujący sposób:

1. Na początku algorytmu, tworzony jest labirynt, którego wszystkie pola stanowią ściany. Rysunek 20 przedstawia początkowy układ labiryntu.



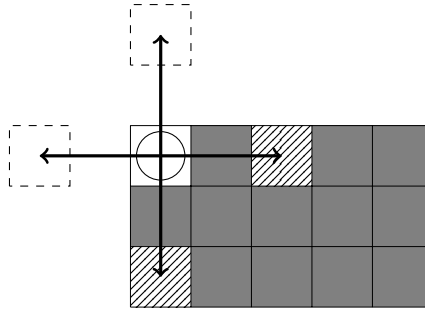
Rysunek 20: Cała plansza stanowi ściany.

2. W następnym kroku wybierana jest losowa komórka, która jest oznaczana jako przejście. Na rysunku 21 pokazano losowo wybrane pole labiryntu.



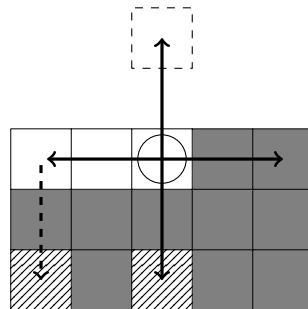
Rysunek 21: Losowo wybrane pole, zamienione na przejście.

3. Następnie algorytm losowo wybiera jedną z sąsiadujących komórek. Podobnie jak w przypadku *algorytmu Prima*, przyjęto, że komórka sąsiadująca to dowolna komórka oddalona o jedno pole w pionie lub poziomie. Podczas ruchu przejścia są weryfikowane - mogą bowiem znajdować się poza granicami planszy. Rysunek 22 przedstawia taką sytuację.

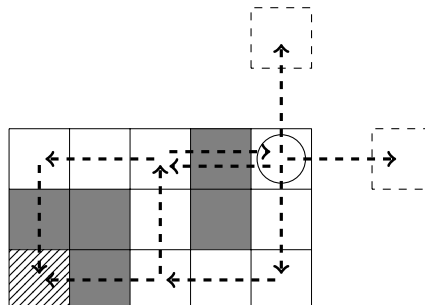


Rysunek 22: Potencjalne dalsze ruchy algorytmu (oznaczone strzałkami). Nieistniejące pola (poza labiryntem) oznaczono przerywaną linią, natomiast pola stanowiące rzeczywiste możliwe przejścia zakreskowano. Aktualnie rozpatrywaną komórkę oznaczono kółkiem.

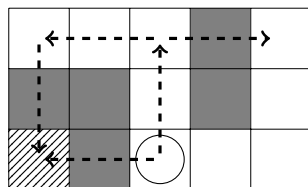
4. Po losowym wybraniu nowej komórki do której można utworzyć ścieżkę, algorytm usuwa ścianę dzielącą ją od poprzedniej i kontynuuje działanie na tej komórce. Warto jednak zauważyć, że ze względu na rekurencyjny charakter algorytmu, poprzednie możliwe ruchy są zapisywane na *stosie*. Oznacza to, że algorytm generuje labirynt *w głąb*, podejmując decyzję o kontynuacji na podstawie pierwszej dostępnej komórki, a dopiero w przypadku braku dalszych możliwości następuje wycofanie do wcześniejszego kroku. Rysunek 23 ilustruje cały ten proces.



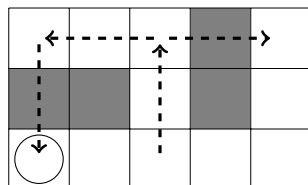
Rysunek 23: Wybór kolejnego pola. Ruchy występujące w poprzednich krokach, ale nie brane pod uwagę w bieżącej iteracji oznaczono przerywaną linią.



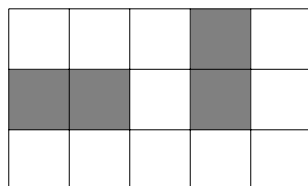
Rysunek 24: Stan labiryntu po kilku następnych iteracjach. Algorytm zacznie wracać, gdyż nie istnieje żaden prawidłowy ruch generujący przejście - ruchy w górę lub prawo spowodują wygenerowanie przejścia poza planszę, a ruchy w lewo bądź dół - powstanie pętli.



Rysunek 25: Odnalezienie prawidłowego przejścia przy powrocie.



Rysunek 26: Wylosowanie kierunku tworzącego przejście.



Rysunek 27: Końcowy wygląd labiryntu. Pozostałe powroty nie wygenerowały żadnych przejść ze względu na brak prawidłowych ruchów.

3 Wyszukiwanie drogi

3.1 Stany węzłów

Rozróżniamy następujące stany węzłów:

- *Selected* – pole zaklasyfikowane przez algorytm jako element końcowej ścieżki.
- *Candidate* – pole aktualnie analizowane przez algorytm.
- *Queued* – pole dodane do kolejki przetwarzania, które wkrótce stanie się polem *Candidate*.
- *Forsaken* – pole wcześniej rozważane jako *Candidate*, lecz odrzucone w dalszym przebiegu algorytmu.

Oznaczenia węzłów oraz ich stanów przedstawia rysunek 28.



(a) Šciana.



(b) Przejście.



(c) Pole *candidate*.



(d) Koniec ścieżki.



(e) Pole startowe.



(f) Pole *selected*.



(g) Pole *forsaken*.



(h) Pole *queued*.

Rysunek 28: Oznaczenia węzłów.

3.2 Algorytm DFS

3.2.1 Opis działania algorytmu

Algorytm wykorzystuje **przeszukiwanie w głąb (Depth-First Search)** do znalezienia ścieżki między punktem startowym a końcowym w labiryncie. Poniżej przedstawiono kluczowe kroki działania.

3.2.2 Inicjalizacja

1. Inicjalizacja struktur danych:

- **stack** - stos przechowujący węzły do odwiedzenia (zainicjowany pozycją startową),
- **state** - mapa śledząca stan węzłów (czy węzeł został odwiedzony).

2. Oznaczenie węzła startowego jako **queued** (w kolejce).

3.2.3 Funkcja rekurencyjna DFS

Główna logika zaimplementowana jest w funkcji rekurencyjnej `dfs(currentNodePos)` przedstawia algorytm 1.

Algorytm 1 Procedura DFS

Oznacz bieżący węzeł jako **candidate** (kandydat)

Jeśli bieżący węzeł jest metą (**finish**) **wtedy**

Oznacz węzeł jako **selected** (wybrany)

Zwróć ścieżka zawierająca tylko bieżący węzeł

koniec jeśli

Dla każdego sąsiada (**neighbour**) bieżącego węzła **wykonaj**

Jeśli sąsiad nie jest kolizją i nie był odwiedzony (**candidate/forsaken**) **wtedy**

Oznacz sąsiada jako **queued**

$\text{path} \leftarrow \text{dfs}(\text{pozycja sąsiada})$

Jeśli ścieżka nie jest pusta **wtedy**

Oznacz bieżący węzeł jako **selected**

Dodaj bieżący węzeł do ścieżki

Zwróć ścieżka

w przeciwnym razie

Oznacz sąsiada jako **forsaken** (porzucony)

koniec jeśli

koniec jeśli

koniec dla

Zwróć pusta ścieżka

3.2.4 Proces budowania ścieżki

1. **Propagacja w górę:** Po znalezieniu mety:

- Następuje rekurencyjna propagacja w górę drzewa.
- Każdy węzeł dodaje swoją pozycję do ścieżki.

- Węzły na ścieżce oznaczane są jako **selected**.
2. **Weryfikacja ścieżki:** Jeśli ścieżka nie istnieje, zwracana jest pusta lista.
 3. **Końcowe przetwarzanie:** Po zakończeniu rekurencji ścieżka jest odwracana gdyż budowana jest od mety do startu, a wynik powinien przedstawiać kolejność od startu do mety.

3.2.5 Możliwe stany węzłów

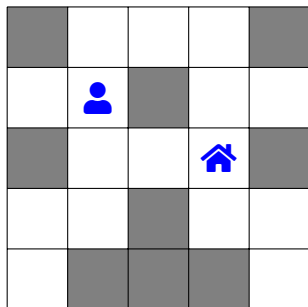
- **queued** - w kolejce do odwiedzenia,
- **candidate** - aktualnie przetwarzany,
- **selected** - część finalnej ścieżki,
- **forsaken** - porzucony (nie prowadzi do mety).

3.2.6 Złożoność obliczeniowa

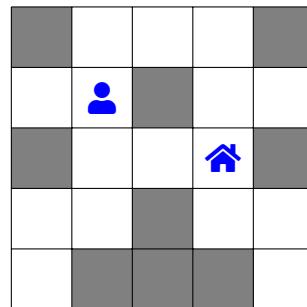
- **Czasowa:** $O(V + E)$, gdzie:
 - V - liczba węzłów,
 - E - liczba krawędzi.
- **Pamięciowa:** $O(V)$, zdeterminowana głębokością rekurencji

Przykład działania algorytmu przedstawiają rysunki 29-49.

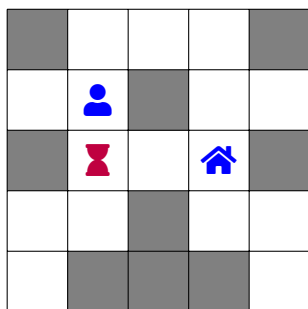
W opisach rysunków współrzędne (x, y) oznaczają: x – kolumna, y – wiersz, licząc od zera.



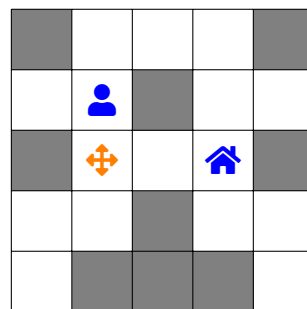
Rysunek 29: Wybierz pole (1,3) jako następnie rozpatywany węzeł.



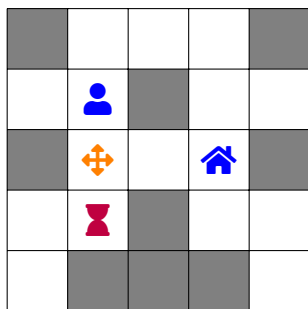
Rysunek 30: Oznacz pole (1,3) jako kandydata na ścieżkę.



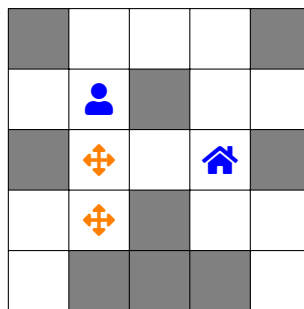
Rysunek 31: Wybierz pole (1,2) jako następnie rozpatywany węzeł.



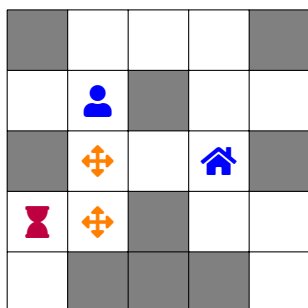
Rysunek 32: Oznacz pole (1,2) jako kandydata na ścieżkę.



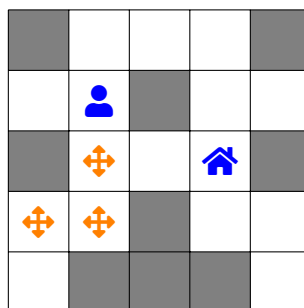
Rysunek 33: Wybierz pole (1,1) jako następnie rozpatywany węzeł.



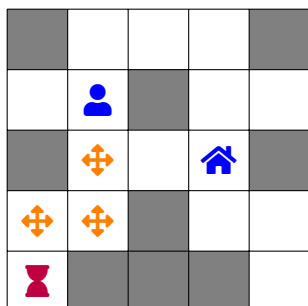
Rysunek 34: Oznacz pole (1,1) jako kandydata na ścieżkę.



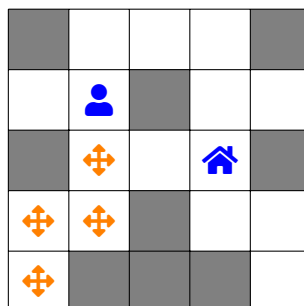
Rysunek 35: Wybierz pole (0,1) jako następnie rozpatywany węzeł.



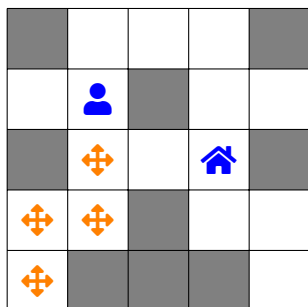
Rysunek 36: Oznacz pole (0,1) jako kandydata na ścieżkę.



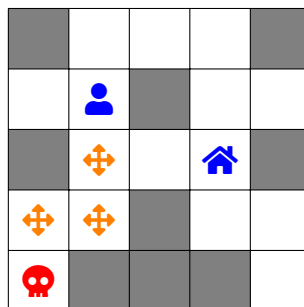
Rysunek 37: Wybierz pole (0,0) jako następnie rozpatywany węzeł.



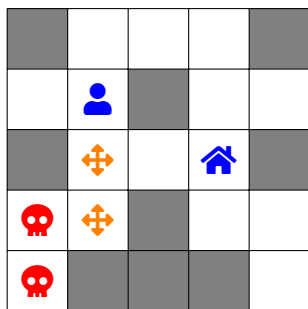
Rysunek 38: Oznacz pole (0,0) jako kandydata na ścieżkę.



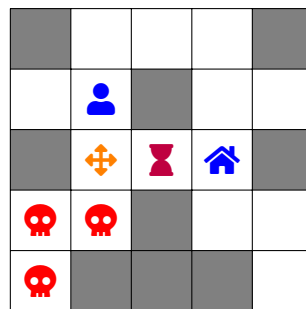
Rysunek 39: Oznacz pole (0,0) jako zapomniany węzeł.



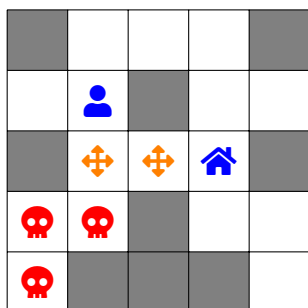
Rysunek 40: Oznacz pole (0,1) jako zapomniany węzeł.



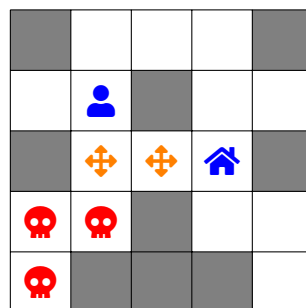
Rysunek 41: Oznacz pole (1,1) jako zapomniany węzeł węzeł.



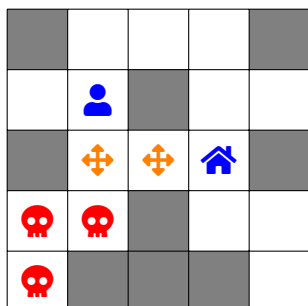
Rysunek 42: Wybierz pole (2,2) jako następnie rozpatrywany węzeł.



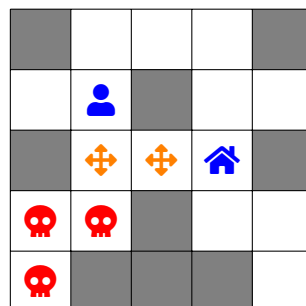
Rysunek 43: Oznacz pole (2,2) jako kandydata na ścieżkę.



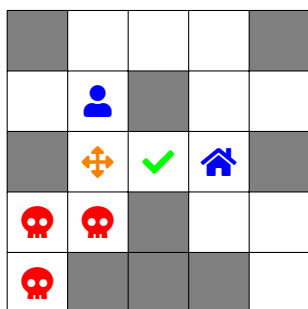
Rysunek 44: Wybierz pole (3,2) jako następnie rozpatrywany węzeł.



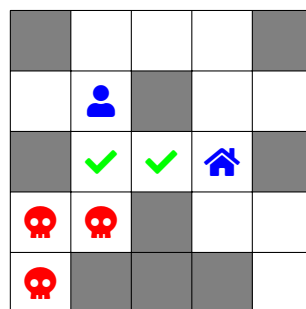
Rysunek 45: Oznacz pole (3,2) jako kandydata na ścieżkę.



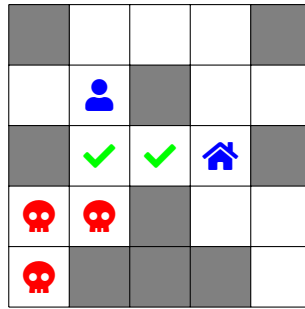
Rysunek 46: Wybierz pole (3,2) do finalnej ścieżki.



Rysunek 47: Wybierz pole (2,2) do finalnej ścieżki.



Rysunek 48: Wybierz pole (1,2) do finalnej ścieżki.



Rysunek 49: Wybierz pole (1,3) do finalnej ścieżki.

3.3 Algorytm BFS

3.3.1 Opis działania algorytmu

Algorytm wykorzystuje **przeszukiwanie wszerz (Breadth-First Search)** do znalezienia najkrótszej ścieżki między punktem startowym a końcowym w labiryncie. Poniżej przedstawiono kluczowe kroki działania:

3.3.2 Inicjalizacja

1. Inicjalizacja struktur danych:
 - **queue** - kolejka FIFO przechowująca węzły do odwiedzenia (zainicjowana pozycją startową),
 - **state** - mapa śledząca pochodzenie węzłów (klucz: pozycja, wartość: pozycja rodzica).
2. Oznaczenie węzła startowego jako **queued** (w kolejce).

3.3.3 Proces przeszukiwania BFS

Główna logika zaimplementowana jest w pętli przetwarzającej kolejkę, co przedstawiono w algorytmie 2:

3.3.4 Proces budowania ścieżki

1. Jeśli cel (*end*) został osiągnięty:
 - Inicjalizacja pustej ścieżki.
 - Backtracking od celu do startu:
 - (a) Dodaj aktualną pozycję do ścieżki.
 - (b) Przejdź do pozycji rodzica (z mapy **state**).
 - (c) Oznacz węzeł jako **selected**.
 - Oznaczenie węzła startowego jako **selected**.
 - Odwrócenie ścieżki (od startu do celu).
2. Jeśli cel nie został osiągnięty, zwracana jest pusta lista.

Algorytm 2 Procedura BFS

```
Dopóki kolejka nie jest pusta wykonaj
  current ← queue.dequeue()
  Oznacz current jako candidate (kandydat)
  Jeśli current = end wtedy
    break
  koniec jeśli
  Dla każdego sąsiada neighbour bieżącego węzła wykonaj
    Jeśli neighbour nie jest ścianą i nie był odwiedzony wtedy
      state[neighbour] ← current (zapisz pochodzenie)
      queue.enqueue(neighbour)
      Oznacz neighbour jako queued
    koniec jeśli
  koniec dla
koniec dopóki
```

3.3.5 Złożoność obliczeniowa

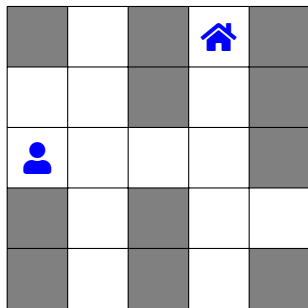
- **Czasowa:** $O(V + E)$, gdzie:
 - V - liczba wierzchołków (komórek labiryntu),
 - E - liczba krawędzi (połączeń między komórkami).
- **Pamięciowa:** $O(V)$:
 - Przechowywanie odwiedzonych węzłów w mapie stanu.
 - Kolejka przechowująca do $O(V)$ elementów.

3.3.6 Właściwości algorytmu

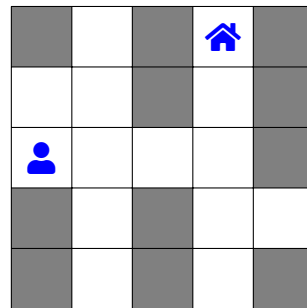
- Gwarantuje znalezienie **najkrótszej ścieżki** (w sensie liczby kroków).
- Eksploruje równomiernie we wszystkich kierunkach.
- Wymaga **pełnej eksploracji** w najgorszym przypadku.

Przykład działania algorytmu przedstawiają rysunki 50-83.

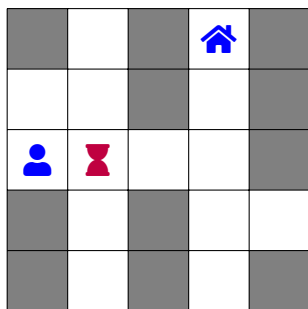
W opisach rysunków współrzędne (x, y) oznaczają: x - kolumna, y - wiersz, licząc od zera.



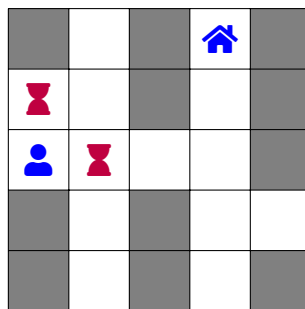
Rysunek 50: Dodaj do kolejki węzeł (0,2).



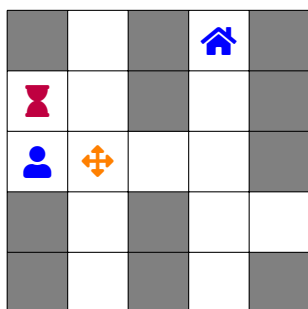
Rysunek 51: Rozpatrz pole (0,2).



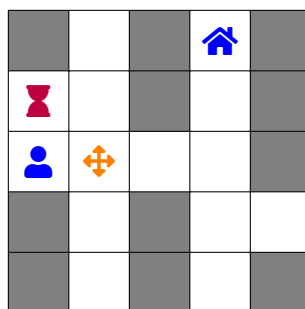
Rysunek 52: Dodaj do kolejki węzeł (1,2).



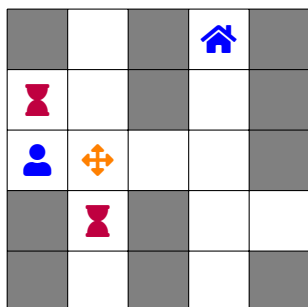
Rysunek 53: Dodaj do kolejki węzeł (0,3).



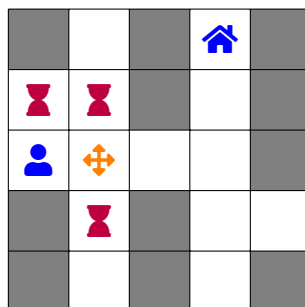
Rysunek 54: Rozpatrz pole (1,2).



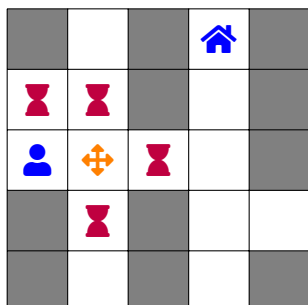
Rysunek 55: Dodaj do kolejki węzeł (0,2).



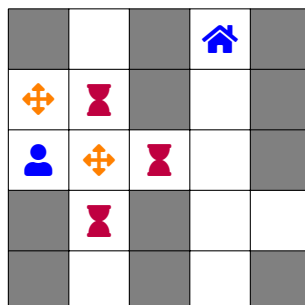
Rysunek 56: Dodaj do kolejki węzeł (1,1).



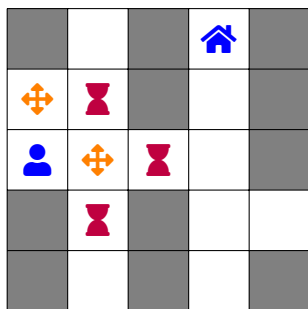
Rysunek 57: Dodaj do kolejki węzeł (1,3).



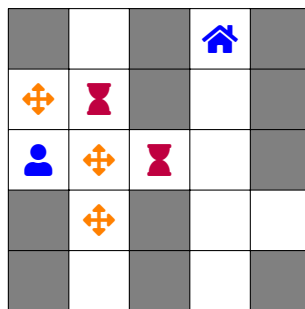
Rysunek 58: Dodaj do kolejki węzeł (2,2).



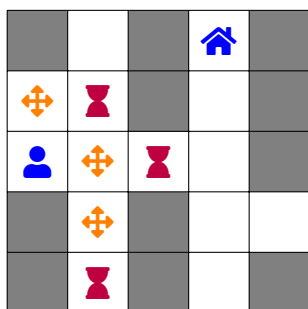
Rysunek 59: Rozpatrz pole (0,3).



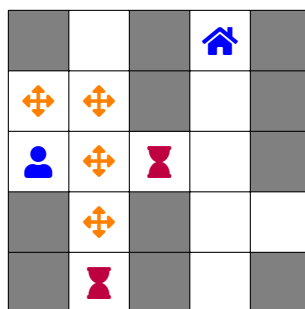
Rysunek 60: Rozpatrz pole (0,2).



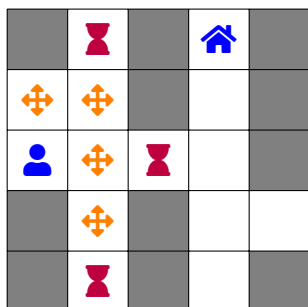
Rysunek 61: Rozpatrz pole (1,1).



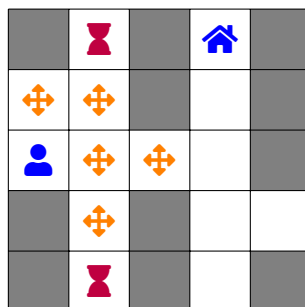
Rysunek 62: Dodaj do kolejki węzeł (1,0).



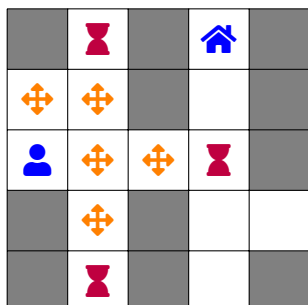
Rysunek 63: Rozpatrz pole (1,3).



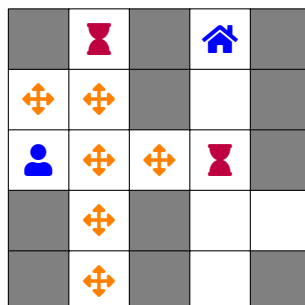
Rysunek 64: Dodaj do kolejki węzeł (1,4).



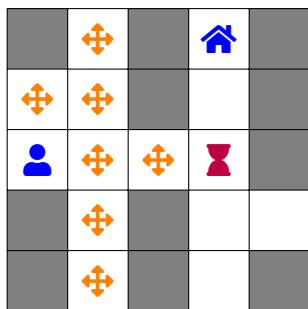
Rysunek 65: Rozpatrz pole (2,2).



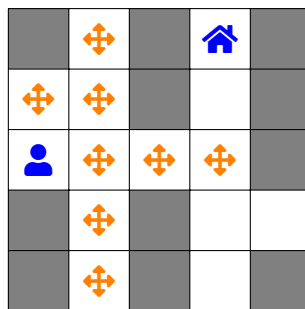
Rysunek 66: Dodaj do kolejki węzeł (3,2).



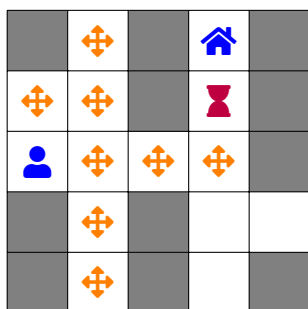
Rysunek 67: Rozpatrz pole (1,0).



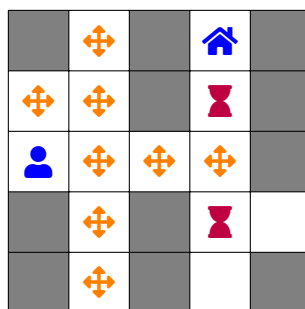
Rysunek 68: Rozpatrz pole (1,4).



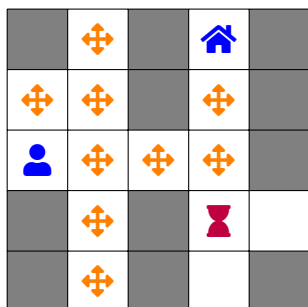
Rysunek 69: Rozpatrz pole (3,2).



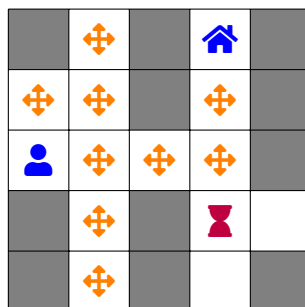
Rysunek 70: Dodaj do kolejki węzeł (3,3).



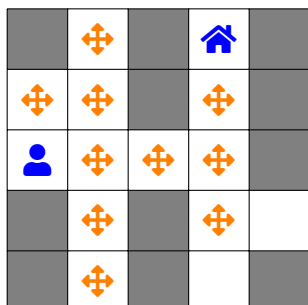
Rysunek 71: Dodaj do kolejki węzeł (3,1).



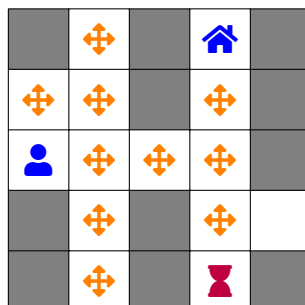
Rysunek 72: Rozpatrz pole (3,3).



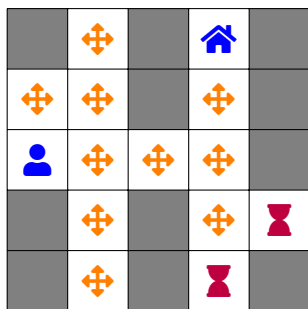
Rysunek 73: Dodaj do kolejki węzeł (3,4).



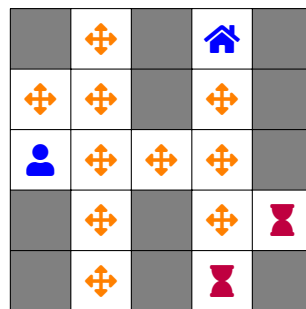
Rysunek 74: Rozpatrz pole (3,1).



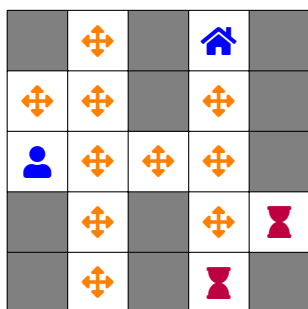
Rysunek 75: Dodaj do kolejki węzeł (3,0).



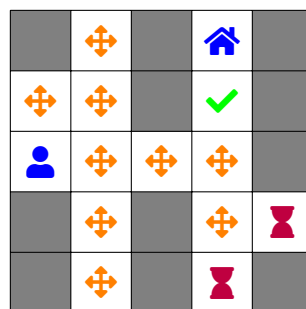
Rysunek 76: Dodaj do kolejki węzeł (4,1).



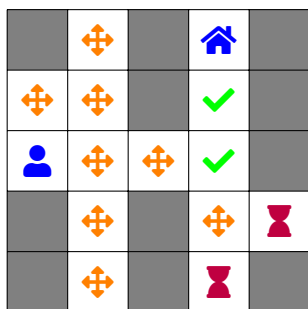
Rysunek 77: Rozpatrz pole (3,4).



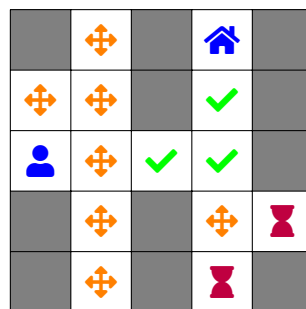
Rysunek 78: Wybierz (3,4) do finalnej ścieżki.



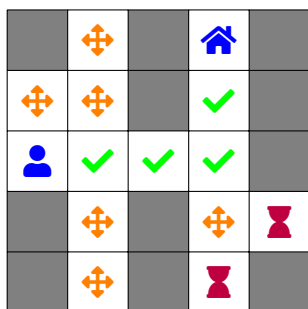
Rysunek 79: Wybierz (3,3) do finalnej ścieżki.



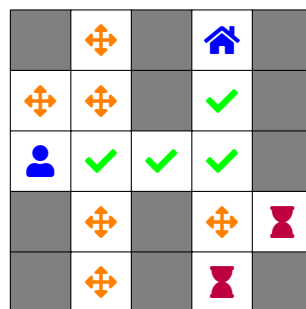
Rysunek 80: Wybierz (3,2) do finalnej ścieżki.



Rysunek 81: Wybierz (2,2) do finalnej ścieżki.



Rysunek 82: Wybierz (1,2) do finalnej ścieżki.



Rysunek 83: Wybierz (0,2) do finalnej ścieżki.

3.4 Algorytm A*

Algorytm A* jest algorytmem wyszukiwania ścieżki w grafie, który znajduje najkrótszą ścieżkę między punktem startowym a docelowym. Wykorzystuje funkcję heurystyczną do optymalizacji procesu przeszukiwania.

3.4.1 Opis działania algorytmu

Algorytm łączy zalety przeszukiwania wszerek (BFS) i zachłannego przeszukiwania najlepszego pierwszego (Best-First Search). Działa poprzez minimalizację funkcji kosztu (równanie (1)).

$$f(n) = g(n) + h(n) \quad (1)$$

gdzie:

- $g(n)$ - rzeczywisty koszt dotarcia z węzła startowego do bieżącego,
- $h(n)$ - heurystyczny koszt dotarcia z bieżącego węzła do celu.

3.4.2 Inicjalizacja

1. Inicjalizacja struktur danych:

- **state** - mapa stanów węzłów (przechowuje $g(n)$ i poprzednika),
- **sortedQueue** - kolejka priorytetowa węzłów (posortowana po $f(n)$).

2. Dodanie węzła startowego:

- $g(\text{start}) = 0$
- $f(\text{start}) = h(\text{start})$
- Oznaczenie startu jako **queued**.

3.4.3 Funkcja heurystyczna

Wykorzystana heurystyka to **odległość Manhattan (Taxicab)** przedstawiona w równaniu 2:

$$h(n) = |n_x - \text{end}_x| + |n_y - \text{end}_y| \quad (2)$$

Zapewnia dopuszczalność (nigdy nie oszacuje, że trzeba przejść więcej kroków niż faktycznie potrzeba).

3.4.4 Główna pętla algorytmu

Główna pętla algorytmu A* została przedstawiona w algorytmie 3.

Algorytm 3 Główna pętla algorytmu A*

Dopóki kolejka nie jest pusta **wykonaj**
 Pobierz węzeł o minimalnym $f(n)$ z `sortedQueue`
 Oznacz bieżący węzeł jako `candidate`
 Jeśli bieżący węzeł jest metą **wtedy**
 Przerwij pętlę
 koniec jeśli
 Dla każdego sąsiada **wykonaj**
 Jeśli sąsiad nie jest kolizją i nie był odwiedzony **wtedy**
 $g_{\text{new}} \leftarrow g(\text{current}) + 1$
 $f_{\text{new}} \leftarrow g_{\text{new}} + h(\text{sąsiad})$
 Zapisz stan: $g = g_{\text{new}}$, poprzednik = `currentNodePos`
 Wstaw do kolejki z priorytetem f_{new}
 Oznacz jako `queued`
 koniec jeśli
 koniec dla
koniec dopóki

3.4.5 Budowanie ścieżki

1. Śledzenie wsteczne:

- Rozpocznij od mety.
- Podążaj do poprzedników aż do startu.
- Oznaczaj węzły jako `selected`.

2. Końcowe przetwarzanie:

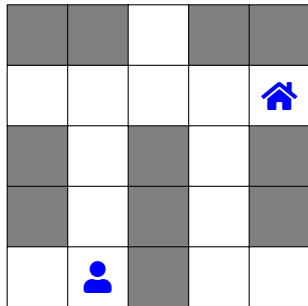
- Odwróć ścieżkę ($\text{start} \rightarrow \text{meta}$).
- Oznacz start jako `selected`.

3.4.6 Złożoność obliczeniowa

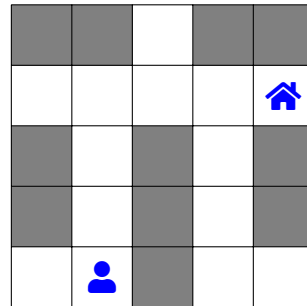
- **Czasowa:** $O(n^2)$ - dla implementacji z listą.
- **Pamięciowa:** $O(n)$ - przechowywanie stanów i kolejki.

Przykład działania algorytmu przedstawiają rysunki 84-108.

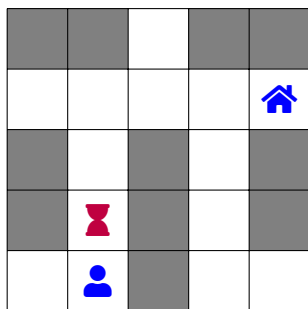
W opisach rysunków współrzędne (x, y) oznaczają: x – kolumna, y – wiersz, licząc od zera.



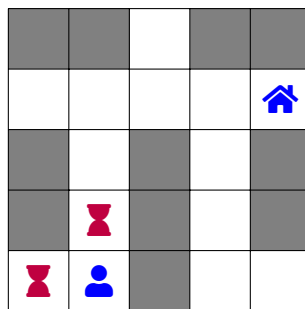
Rysunek 84: Dodaj do kolejki węzeł (1,0).



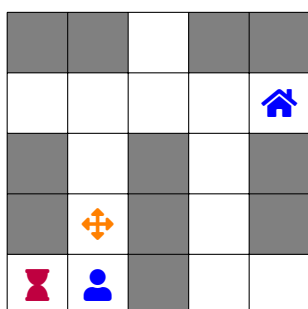
Rysunek 85: Rozpatrz pole (1,0).



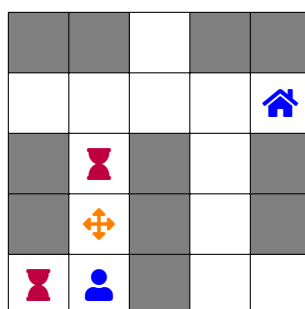
Rysunek 86: Dodaj do kolejki węzeł (1,1).



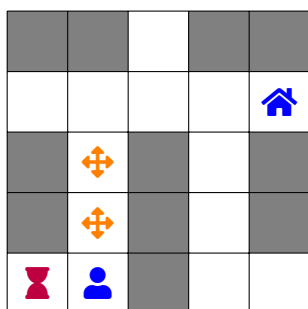
Rysunek 87: Dodaj do kolejki węzeł (0,0).



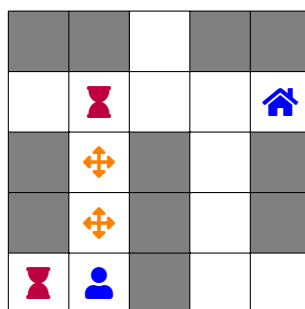
Rysunek 88: Rozpatrz pole (1,1).



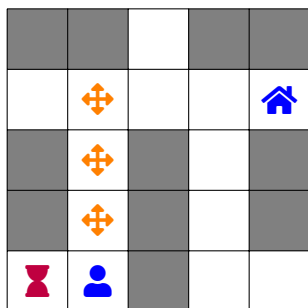
Rysunek 89: Dodaj do kolejki węzeł (1,2).



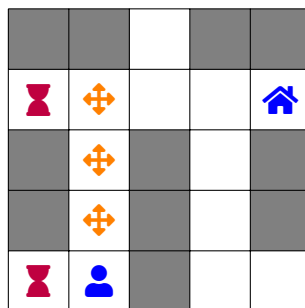
Rysunek 90: Rozpatrz pole (1,2).



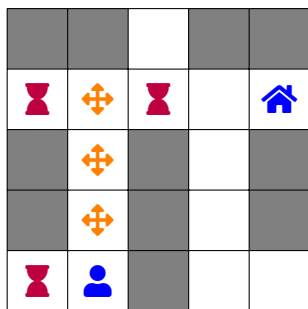
Rysunek 91: Dodaj do kolejki węzeł (1,3).



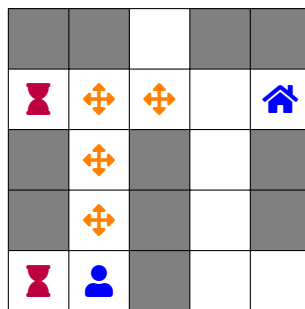
Rysunek 92: Rozpatrz pole (1,3).



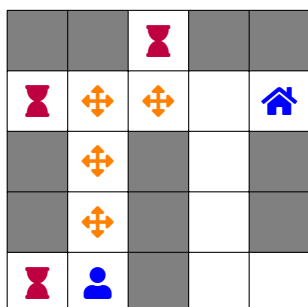
Rysunek 93: Dodaj do kolejki węzeł (0,3).



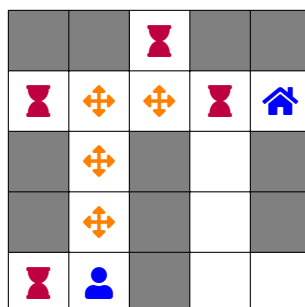
Rysunek 94: Dodaj do kolejki węzeł (2,3).



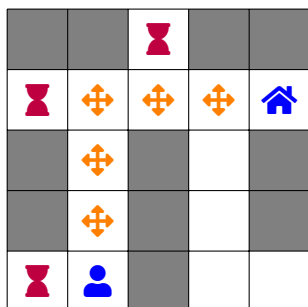
Rysunek 95: Rozpatrz pole (2,3).



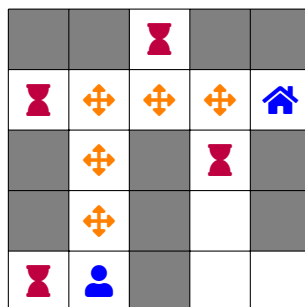
Rysunek 96: Dodaj do kolejki węzeł (2,4).



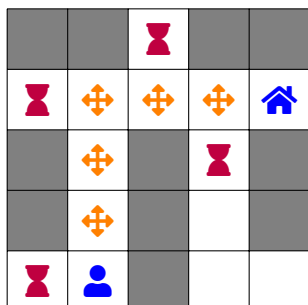
Rysunek 97: Dodaj do kolejki węzeł (3,3).



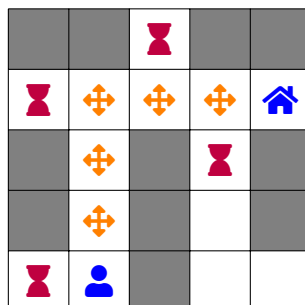
Rysunek 98: Rozpatrz pole (3,3).



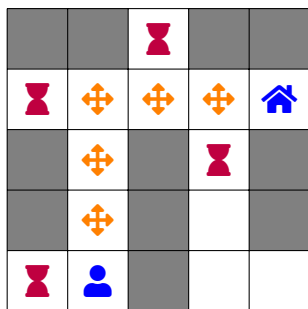
Rysunek 99: Dodaj do kolejki węzeł (3,2).



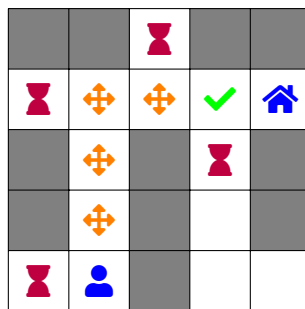
Rysunek 100: Dodaj do kolejki węzeł (4,3).



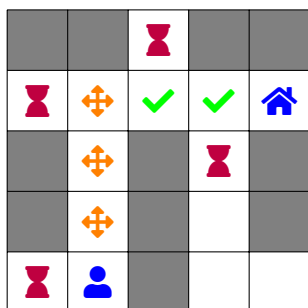
Rysunek 101: Rozpatrz pole (4,3).



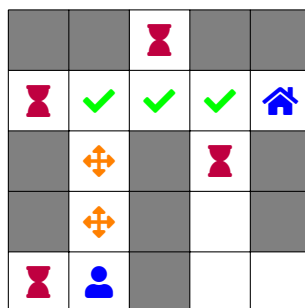
Rysunek 102: Wybierz (4,3) do finalnej ścieżki.



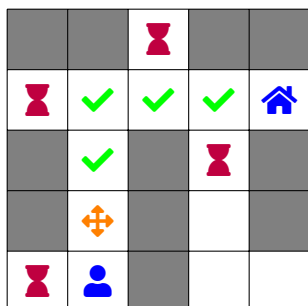
Rysunek 103: Wybierz (3,3) do finalnej ścieżki.



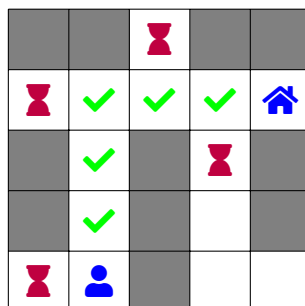
Rysunek 104: Wybierz (2,3) do finalnej ścieżki.



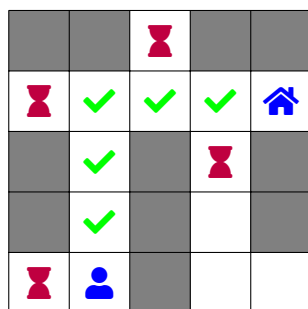
Rysunek 105: Wybierz (1,3) do finalnej ścieżki.



Rysunek 106: Wybierz (1,2) do finalnej ścieżki.



Rysunek 107: Wybierz (1,1) do finalnej ścieżki.

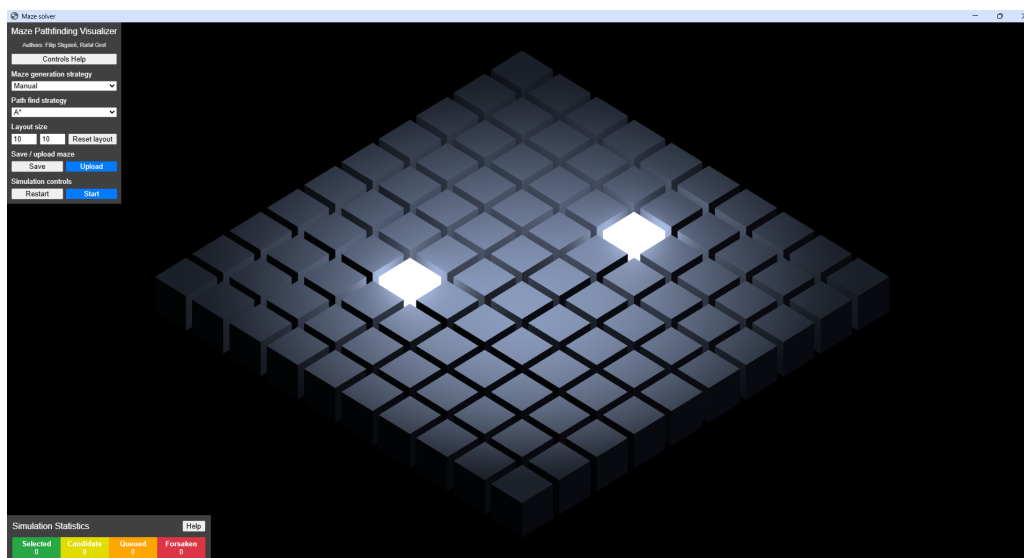


Rysunek 108: Wybierz (1,0) do finalnej ścieżki.

4 Instrukcja obsługi aplikacji

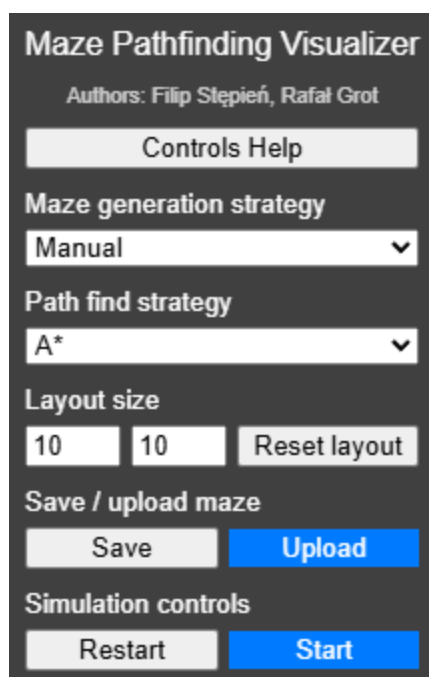
4.1 Uruchamianie

Aby uruchomić aplikację, należy otworzyć plik z rozszerzeniem `*.html` (`index.html`) przy pomocy dowolnej przeglądarki. Po uruchomieniu interfejs użytkownika powinien wyglądać jak na rysunku 109.



Rysunek 109: Uruchomiona aplikacja.

4.2 Interfejs użytkownika



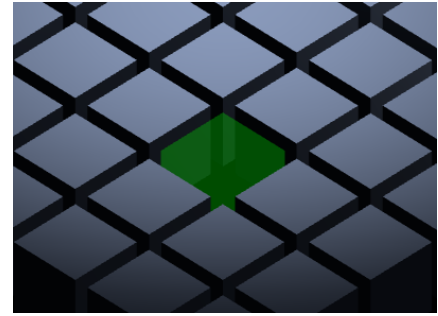
Rysunek 110: Panel sterowania.

Panel sterowania (rysunek 110) zawiera wszystkie kluczowe opcje do manipulacji labiryntem:

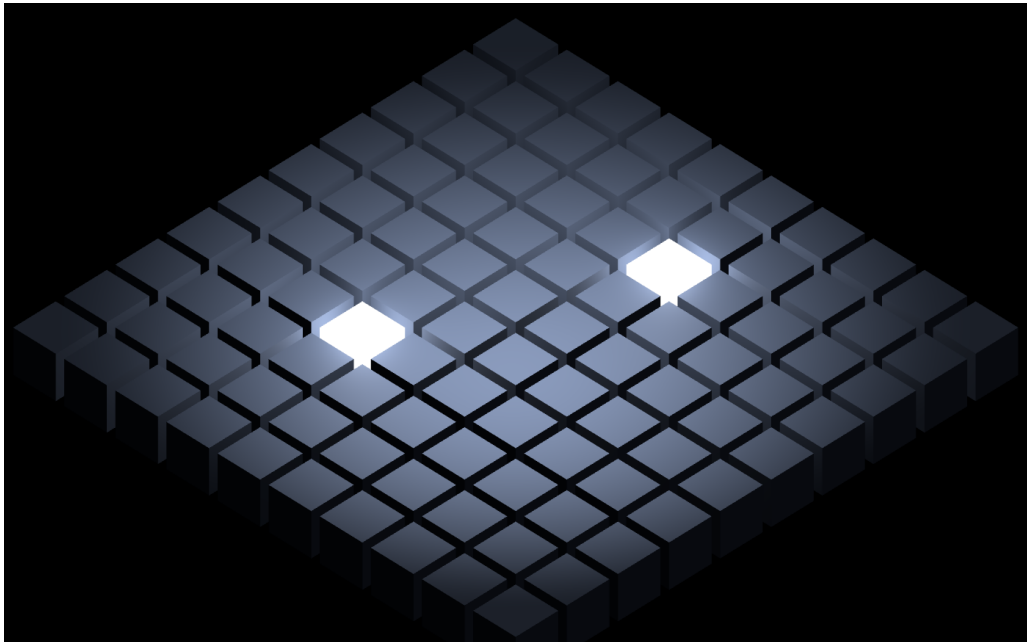
- *Controls Help* – przycisk pokazujący skróty klawiszowe.
- *Maze generation strategy* – wybór algorytmu generowania labiryntu; opcja *Manual* pozwala ręcznie stworzyć planszę.
- *Path find strategy* – wybór algorytmu do znalezienia ścieżki.
- *Layout size* – wybór szerokości i wysokości planszy. *Reset layout* przywróci planszę do stanu początkowego.
- *Save / upload maze* – pobieranie lub wczytywanie labiryntu z pliku.
- *Simulation controls* – sterowanie symulacją: *Restart* usuwa znaczniki wyszukanej drogi, *Start* rozpoczyna wyszukiwanie drogi.

Plansza labiryntu (rysunek 112) stanowi główny element wizualizacji. Składa się z symetrycznych sześciątów — obecność sześcianu oznacza możliwą **drogę**, jego brak wskazuje na **ścianę**. Użytkownik może najechać kursorem na dowolne pole niebędące punktem startowym ani końcowym (oznaczone kolorem białym), co skutkuje jego podświetleniem na zielono (rysunek 111). Kliknięcie **lewym przyciskiem myszy** usuwa drogę, natomiast **prawym** ją przywraca.

Możliwa jest również modyfikacja położenia punktów startowego i końcowego. W tym celu należy przytrzymać klawisz **Shift** i kliknąć **prawym przyciskiem myszy**, aby zmienić pozycję startu, lub przytrzymać **Ctrl** i kliknąć **prawym przyciskiem myszy**, aby ustawić nowy punkt końcowy.



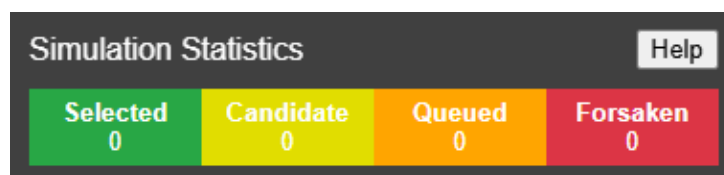
Rysunek 111: Wybranie oraz usunięcie pola labiryntu.



Rysunek 112: Plansza labiryntu.

Panel statystyk (rysunek 113) prezentuje liczbę pól labiryntu przypisanych do poszczególnych kategorii, (patrz: **3.1 Stany węzłów**), które algorytm wykorzystuje podczas poszukiwania ścieżki.

Informacja o znaczeniu poszczególnych statystyk jest zawsze dostępna po kliknięciu przycisku *Help*.



Rysunek 113: Panel statystyk.

Kolory statystyk odpowiadają barwom klasyfikowanych pól w symulacji. Kolorowanie pól labiryntu podczas pracy algorytmu przedstawiono na rysunku 114. Aktualny krok dodatkowo podświetla wybrane pole na odpowiadający mu kolor.



Rysunek 114: Przebieg działania algorytmu. Widoczne są pola startowe i końcowe oznaczone kolorem białym, pola *Candidate* w kolorze żółtym, pola *Forsaken* w kolorze czerwonym oraz pole *Queued* (aktualny krok algorytmu) w kolorze pomarańczowym.

4.3 Przeprowadzanie symulacji

Typowy przebieg czynności po uruchomieniu aplikacji wygląda następująco - wszystkie wymienione kroki wykonuje się w **panelu sterowania**:

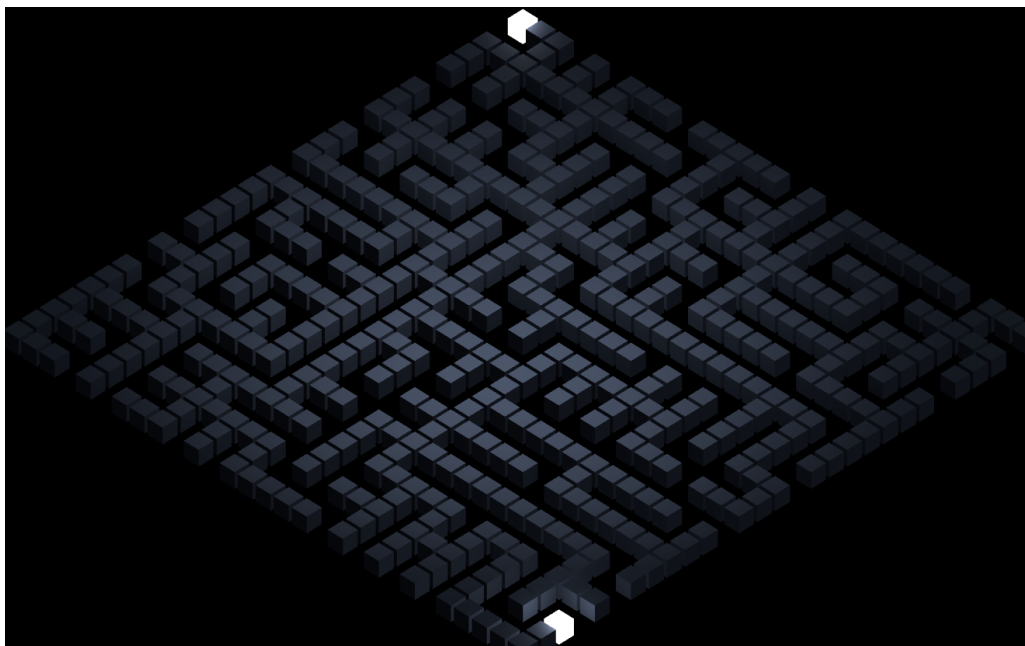
1. Wybranie algorytmu generowania labiryntu (sekcja *Maze generation strategy*).
2. Określenie wielkości planszy (sekcja *Layout size*).
3. Wylosowanie struktury planszy odpowiadającej preferencjom użytkownika (przycisk *Reset layout* w sekcji *Layout size*).
4. Opcjonalne zapisanie struktury labiryntu do pliku do późniejszego wczytania (przycisk *Save* w sekcji *Save/upload maze*).
5. Wybór algorytmu wyszukiwania ścieżki (sekcja *Path find strategy*).
6. Uruchomienie symulacji (przycisk *Start* w sekcji *Simulation controls*).
7. Monitorowanie przebiegu algorytmu oraz analizowanie danych w **panelu statystyk**.

5 Porównanie algorytmów wyszukiwania ścieżki

5.1 Symulacja

Porównanie przeprowadzono na labiryncie o wymiarach 30×30 , wygenerowanym z wykorzystaniem **algorytmu Prima**. Punkty początkowy i końcowy zostały umieszczone odpowiednio na górnej i dolnej krawędzi planszy, co wymusza przejście przez całą jej wysokość. Strukturę wygenerowanego labiryntu przedstawiono na rysunku 115.

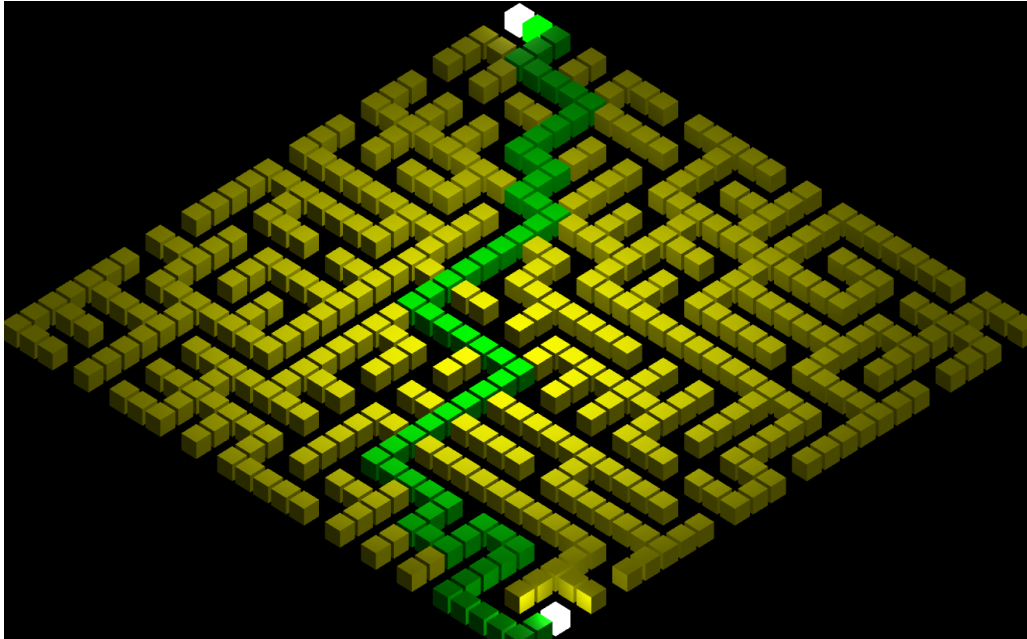
Wyniki działania poszczególnych algorytmów wyszukiwania ścieżki zaprezentowano na rysunkach 116–118 oraz w tabeli 1.



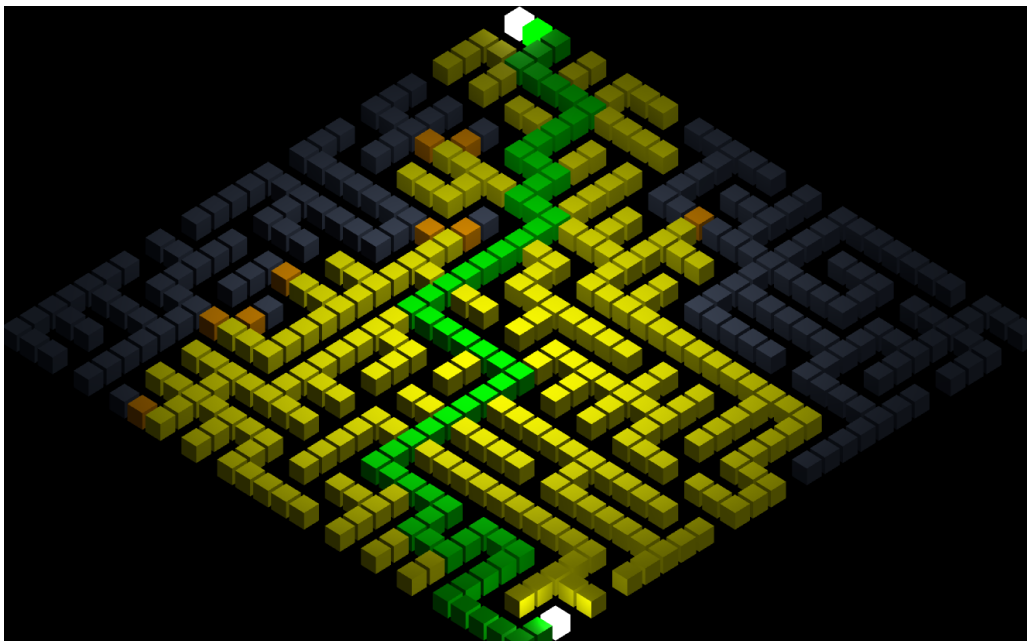
Rysunek 115: Wygenerowany losowo labirynt, użyty w porównaniu.



Rysunek 116: Wynik działania algorytmu DFS.



Rysunek 117: Wynik działania algorytmu BFS.



Rysunek 118: Wynik działania algorytmu A*.

| Algorytm | Ilość pól danej kategorii | | | | Suma kroków |
|----------|---------------------------|------------------|---------------|--------------------|-------------|
| | <i>Selected</i> | <i>Candidate</i> | <i>Queued</i> | <i>Forsaken</i> | |
| DFS | 61 | 197 | 196 | 136 | 590 |
| BFS | 61 | 447 | 446 | <i>nie dotyczy</i> | 954 |
| A* | 61 | 286 | 294 | <i>nie dotyczy</i> | 641 |

Tabela 1: Wyniki działania poszczególnych algorytmów.

5.2 Interpretacja wyników

Analiza danych zawartych w tabeli 1 pozwala na ocenę efektywności i charakterystyki działania poszczególnych algorytmów wyszukiwania ścieżki w kontekście złożoności przestrzeni przeszukiwania oraz liczby wykonanych operacji. Wszystkie trzy algorytmy – *DFS*, *BFS* oraz *A** – znalazły ścieżkę o identycznej długości (61 pól), co potwierdza poprawność ich implementacji oraz jednoznaczność najkrótszego rozwiązania w badanym labiryncie.

Z perspektywy liczby analizowanych pól (*Candidate*) oraz pól dodanych do kolejki (*Queued*), wyraźnie zauważalne są różnice w strategii działania poszczególnych metod. Algorytm *BFS*, przeszukujący przestrzeń wszerz, odznacza się największą liczbą aktywnie analizowanych pól (447) oraz bardzo dużym zasięgiem działania (446 pól w kolejce). Taka strategia zapewnia optymalność rozwiązania, ale wiąże się z wysokim kosztem obliczeniowym.

DFS, jako algorytm oparty na przeszukiwaniu w głąb, wykazuje najniższą łączną liczbę kroków (590), jednak kosztem dużej liczby pól zakwalifikowanych jako *Forsaken* (136). Wynika to z faktu, że *DFS* często „błądzi” w ślepe zaułki, które musi następnie porzucić, co skutkuje koniecznością cofania się i powtórnej eksploracji innych ścieżek.

Algorytm *A**, wykorzystujący heurystykę w ocenie kolejnych pól, stanowi kompromis pomiędzy szerokim przeszukiwaniem a kosztami obliczeniowymi. Choć przetwarza mniej pól niż *BFS*, to nadal zapewnia optymalną ścieżkę. Łączna liczba kroków (641) potwierdza jego wysoką efektywność w kontekście znalezienia najkrótszej trasy przy jednoczesnym ograniczeniu nadmiarowego przeszukiwania.

Podsumowując, algorytm *A** okazał się najbardziej zrównoważony pod względem skuteczności oraz efektywności. *BFS* gwarantuje pełność i optymalność rozwiązania, jednak kosztem znacznie większej liczby operacji. *DFS* natomiast działa szybko i z mniejszym zużyciem zasobów, ale nie daje gwarancji optymalności i może znacząco wydłużyć czas przeszukiwania w bardziej złożonych przypadkach.

6 Zakończenie i wnioski końcowe

Celem niniejszego projektu było zaprojektowanie i implementacja aplikacji umożliwiającej generowanie labiryntów oraz wizualizację działania różnych algorytmów wyszukiwania ścieżek. W toku prac stworzono w pełni funkcjonalne narzędzie, które w sposób interaktywny ilustruje różnice pomiędzy popularnymi algorytmami.

W projekcie zaimplementowano trzy klasyczne algorytmy: DFS (*Depth-First Search*), BFS (*Breadth-First Search*) oraz A*, co umożliwiło ich bezpośrednie porównanie w środowisku symulacyjnym. Każdy z algorytmów został dokładnie przeanalizowany zarówno pod względem teoretycznym, jak i praktycznym, z uwzględnieniem procesu inicjalizacji, budowy ścieżki, stosowanych struktur danych oraz złożoności obliczeniowej.

Przeprowadzone eksperymenty i symulacje wykazały, że zastosowane metody różnią się nie tylko sposobem działania, ale również efektywnością operacyjną. DFS okazał się najoszczędniejszy pod względem liczby operacji, jednak nie gwarantuje optymalności ścieżki. BFS zapewnia zawsze najkrótszą możliwą trasę, lecz obciążone jest to znacznie większym zakresem przeszukiwania. Algorytm A*, wykorzystujący heurystykę w postaci *odległości Manhattan*, zaprezentował najlepszy kompromis pomiędzy dokładnością a efektywnością, co czyni go szczególnie przydatnym w zastosowaniach wymagających wysokiej wydajności.

Na szczególną uwagę zasługuje również stworzony interfejs użytkownika, który w przystępny sposób prezentuje przebieg działania poszczególnych algorytmów. Umożliwia on obserwację procesu wyszukiwania ścieżki w czasie rzeczywistym, a zastosowane oznaczenia kolorystyczne pozwalają na szybkie rozpoznanie aktualnych i przetworzonych węzłów. Funkcjonalność panelu sterowania oraz możliwość zapisu i wczytywania konfiguracji labiryntu znacząco zwiększają użyteczność aplikacji.

Podsumowując, projekt zrealizowano zgodnie z przyjętymi założeniami, zarówno pod względem funkcjonalności, jak i wartości edukacyjnej. Praca nad nim umożliwiła pogłębienie wiedzy z zakresu teorii grafów, algorytmiki oraz projektowania interaktywnych aplikacji. Opracowane rozwiązanie może pełnić funkcję skutecznego wsparcia dydaktycznego w nauczaniu algorytmów przeszukiwania grafów, a także stanowić solidną podstawę do pogłębiania wiedzy na temat analizy i przetwarzania struktur grafowych.

Literatura

- [1] V. K. Balakrishnan. *Schaum's Outline of Theory and Problems of Graph Theory*. McGraw–Hill, New York, nachdr. edition, 2005.
- [2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3rd edition, 2009.