Politechnika Świętokrzyska
Wydział Elektrotechniki, Automatyki i Informatyki

Dokumentacja projektu zespołowego

Wizualizacja wyszukiwania drogi w labiryncie

Filip Stępień — Nr indeksu: 094117
Rafał Grot — Nr indeksu: 094046

Informatyka, grupa 3ID11B

17 czerwca 2025

Spis treści

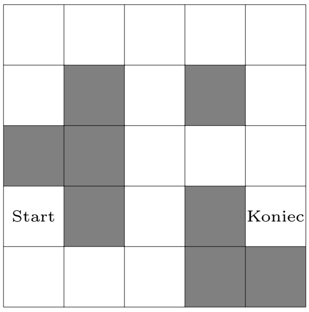
1 Wstęp **3**

2 Generowanie labiryntu **4**
2.1 Algorytm Prima 5
2.2 Algorytm Kruskala 7
2.3 Algorytm *przeszukiwania w głąb* 10

3 Wyszukiwanie drogi **12**
3.1 Algorytm DFS 12
3.1.1 Opis działania algorytmu 12
3.1.2 Inicjalizacja 12
3.1.3 Funkcja rekurencyjna DFS 12
3.1.4 Proces budowania ścieżki 13
3.1.5 Stany węzłów 13
3.1.6 Złożoność obliczeniowa 14
3.2 Algorytm BFS 20
3.2.1 Opis działania algorytmu 20
3.2.2 Inicjalizacja 21
3.2.3 Proces przeszukiwania BFS 21
3.2.4 Proces budowania ścieżki 21
3.2.5 Złożoność obliczeniowa 22
3.2.6 Właściwości algorytmu 22
3.3 Algorytm A* 27
3.4 Algorytm A* 27
3.4.1 Opis działania algorytmu 28
3.4.2 Inicjalizacja 28
3.4.3 Funkcja heurystyczna 28
3.4.4 Główna pętla algorytmu 28
3.4.5 Budowanie ścieżki 29
3.4.6 Złożoność obliczeniowa 29

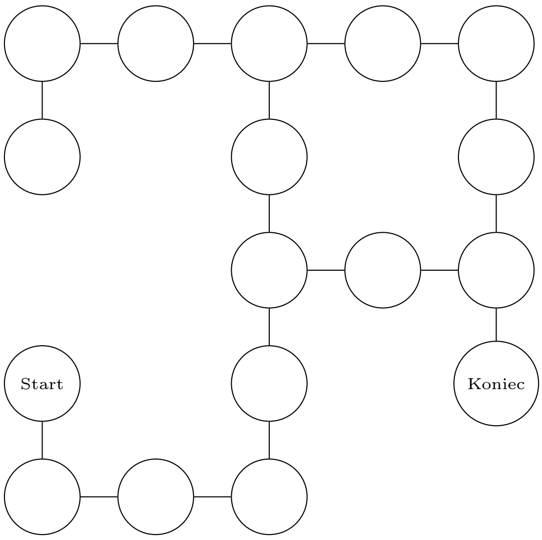
1 Wstęp

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.



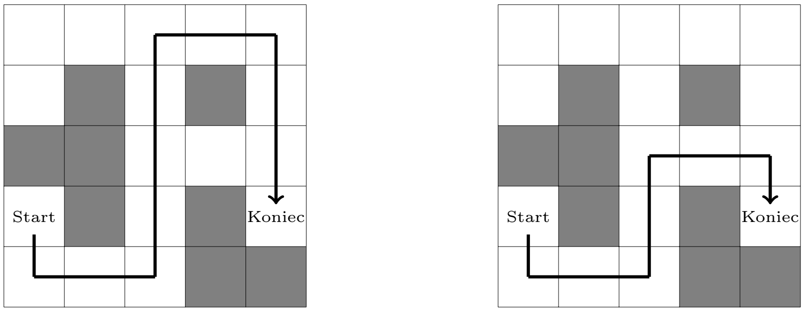
Rysunek 1: Przykładowy labirynt 5x5 z zaznaczonym startem i końcem, gdzie białe pole - przejście, czarne - ściana.

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.



(a) Ścieżka nieoptymalna (b) Ścieżka optymalna

Rysunek 3: Porównanie ścieżek w labiryncie.

2 Generowanie labiryntu

Do generowania labiryntów kluczowe jest zrozumienie idei drzew rozpinających.

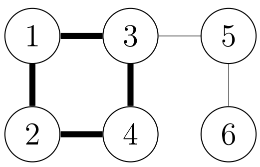
Drzewo rozpinające [2] to taki podgraf oryginalnego grafu, który zawiera wszystkie jego wierzchołki, a jednocześnie jest spójny i nie zawiera cykli.

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.

(a) Graf niespójny. (b) Graf spójny.

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).

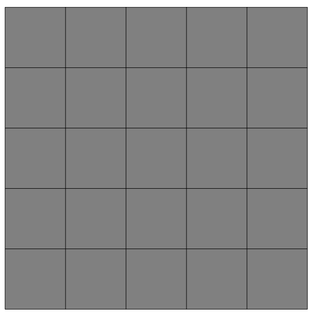
Rysunek 6: Ścieżki spełniające oraz nie spełniające założeń *MST* (pogrubione linie).

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

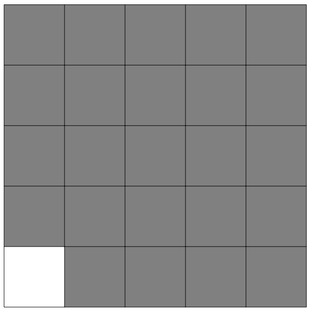
Algorytm Prima [2] w kontekście generowania labiryntu działa poprzez stopniowe budowanie połączeń między komórkami planszy. W każdej iteracji wybierana jest losowa krawędź prowadząca z odwiedzonej komórki do jednej z sąsiednich, jeszcze nieodwiedzonych. Wybrana komórka zostaje następnie połączona z dotychczasowym 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.



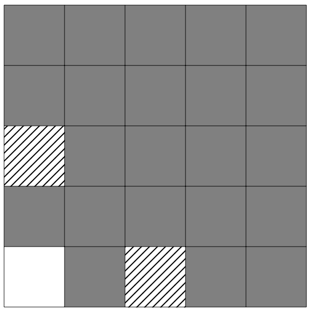
Rysunek 7: Cała plansza stanowi ściany.

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



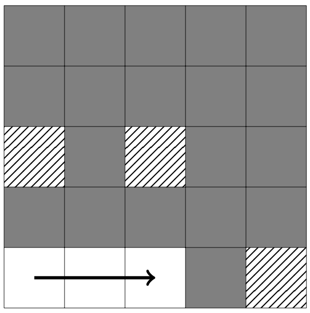
Rysunek 8: Losowe pole startowe.

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.

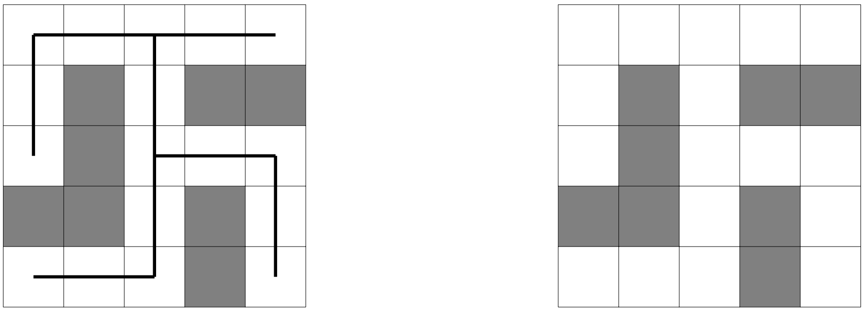
4. Losowana jest jedna krawędź ze zbioru potencjalnych przejść. Jeśli prowadzi ona do nieodwiedzonego pola, tworzy się przejście między bieżącym polem a nowym (usuwana jest ściana między nimi), a nowe pole zostaje oznaczone jako przejście. Następnie do zbioru krawędzi dodawani są sąsiedzi nowo odwiedzonego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

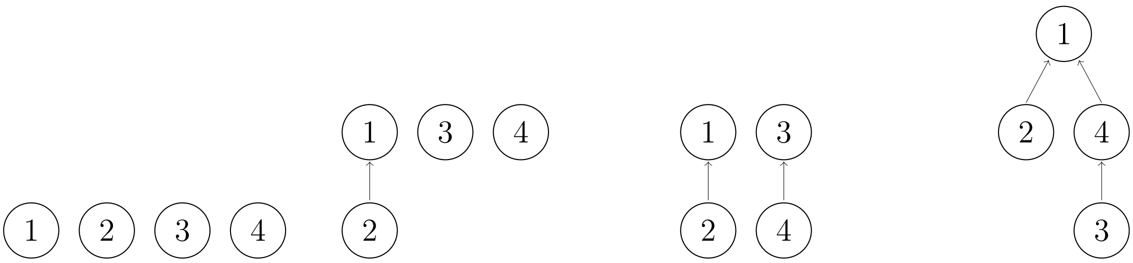
(b) Labirynt powstały na podstawie wyznaczonych krawędzi.

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.



(a) Krok 1 (b) Krok 2 (c) Krok 3 (d) Krok 4

Rysunek 12: Kolejne kroki scalania zbioru

Projekt wykorzystuje zmodyfikowaną wersję algorytmu Kruskala. W klasycznym wariancie 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 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.

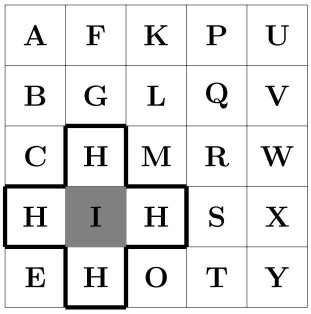
3. Komórki planszy są rozpatrywane w losowej kolejności:

   (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**

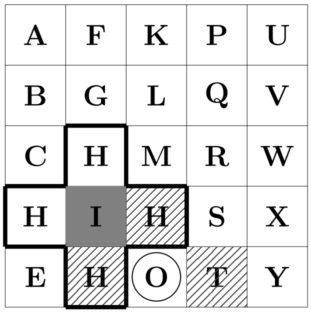
Rysunek 15: Losowa komórka (oznaczona kółkiem) i jej sąsiedzi (zakreskowani).

(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.

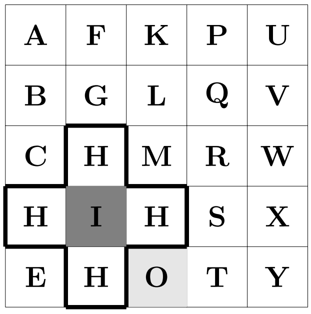


Rysunek 16: Łączenie komórek w zbiory. Reprezentantem zbioru jest pierwszy dodany element (tutaj komórka H), choć może to być dowolna komórka z grupy.

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



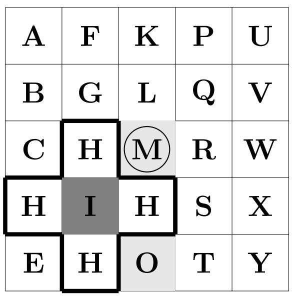
(a) Wylosowanie kolejnej komórki.



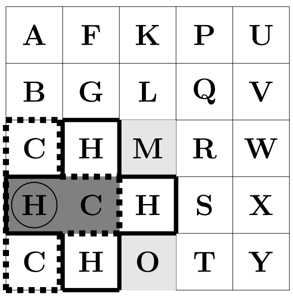
(b) Pozostawienie przejścia.

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

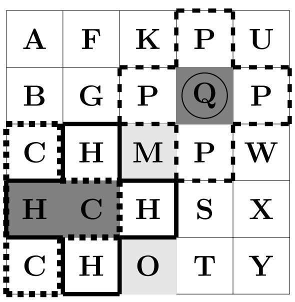
4. Proces powtarza się, aż każda komórka zostanie przetworzona. Przykładowe dalsze kroki algorytmu przedstawia Rysunek 18, a Rysunek 19 ukazuje potencjalny schemat wygenerowanego labiryntu.



(a) Kolejna komórka bez unikalnych sąsiadów.

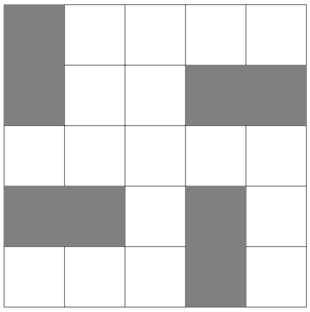


(b) Komórka należąca do scalonego zbioru.



(c) Scalenie kolejnego osobnego zbioru.

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

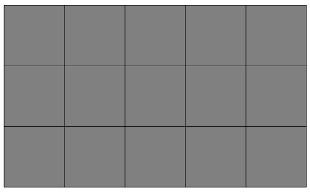


Rysunek 19: Przykładowy wygląd końcowego, wygenerowanego labiryntu.

2.3 Algorytm *przeszukiwania w głąb*

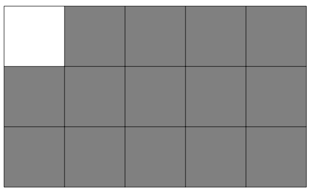
Algorytm przeszukiwania w głąb (ang. *Depth-First Search*, DFS) [2] to klasyczna metoda eksploracji grafu, która w kontekście generowania labiryntu pozwala na systematyczne zagłębianie się w kolejne, jeszcze nieodwiedzone komórki planszy z wykorzystaniem mechanizmu rekurencji. Algorytm rozpoczyna od 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.



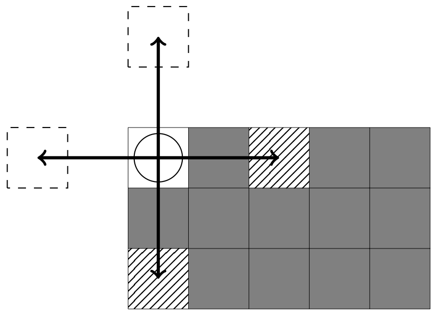
Rysunek 20: Cała plansza stanowi ściany.

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



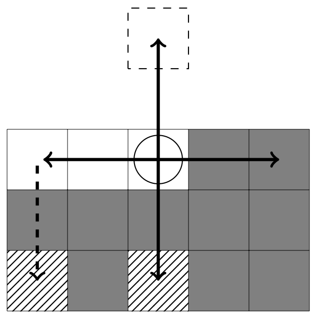
Rysunek 21: Losowo wybrane pole, zamienione na przejście.

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.

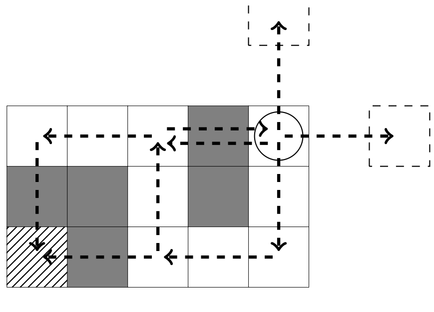


Rysunek 22: Potencjalne dalsze ruchy algorytmu (oznaczone strzałkami). Nieistniejące pola (poza labiryntem) oznaczono przerywaną linią, natomiast pola stanowiące rzeczywiste możliwe przejścia zakreskowano. Aktualnie rozpatrywaną komórkę oznaczono kółkiem.

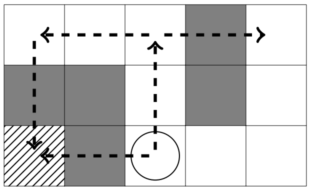
4. Po losowym wybraniu nowej komórki do której można utworzyć ścieżkę, algorytm usuwa ścianę dzielącą ją od poprzedniej i kontynuuje działanie na tej komórce. Warto jednak zauważyć, że ze względu na rekurencyjny charakter algorytmu, poprzednie możliwe ruchy są zapisywane na *stosie*. Oznacza to, że algorytm 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.



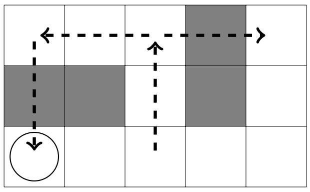
Rysunek 23: Wybór kolejnego pola. Ruchy występujące w poprzednich krokach, ale nie brane pod uwagę w bieżącej iteracji oznaczono przerywaną linią.



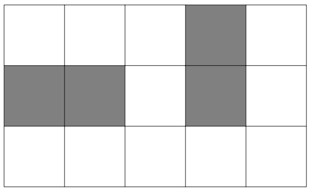
Rysunek 24: Stan labiryntu po kilku następnych iteracjach. Algorytm zacznie wracać, gdyż nie istnieje żaden prawidłowy ruch generujący przejście - ruchy w górę lub prawo spowodują wygenerowanie przejścia poza planszę, a ruchy w lewo bądź dół - powstanie pętli.



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 Algorytm DFS

3.1.1 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:

3.1.2 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)

3.1.3 Funkcja rekurencyjna DFS

Główna logika zaimplementowana jest w funkcji rekurencyjnej `dfs(currentNodePos)`:

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
    path ← dfs(pozycja sasiada)
    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
```

3.1.4 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:

     $$\text{finalna\_ścieżka} = \text{reverse}(\text{ścieżka\_z\_rekurencji})$$

   - Powód: ścieżka budowana jest od mety do startu, a wynik powinien przedstawiać kolejność od startu do mety

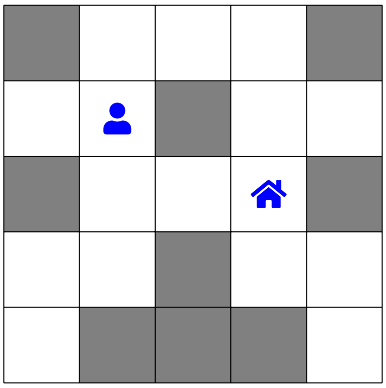
3.1.5 Stany węzłów

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

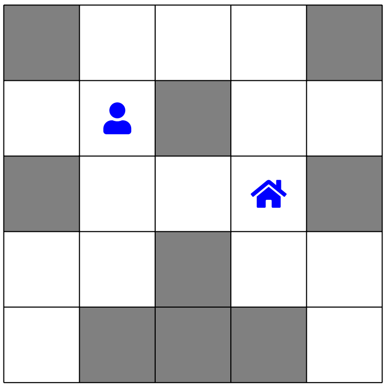
3.1.6 Złożoność obliczeniowa

- **Czasowa**: $O(V + E)$
  - V - liczba węzłów
  - E - liczba krawędzi
- **Pamięciowa**: $O(V)$
  - Zdeterminowana głębokością rekurencji

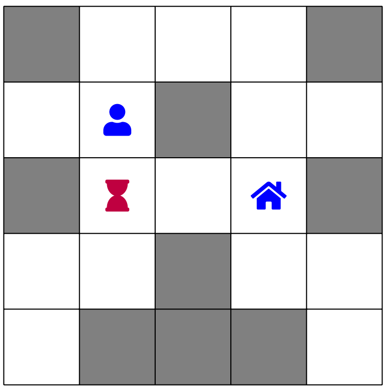
Przykład działania algorytmu przedstawia rysunek 28.



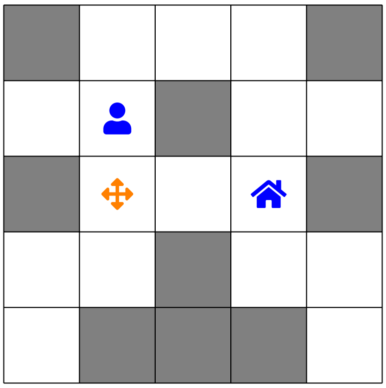
Rysunek 28: Wybierz "x":1,"y":3 jako następnie rozpatywany węzeł



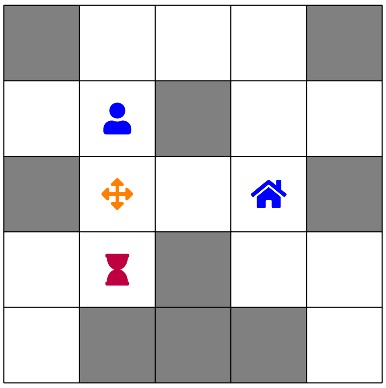
Rysunek 28: Oznacz "x":1,"y":3 jako kandydata



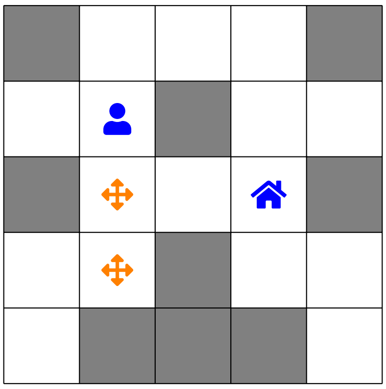
Rysunek 28: Wybierz "x":1,"y":2 jako następnie rozpatywany węzeł



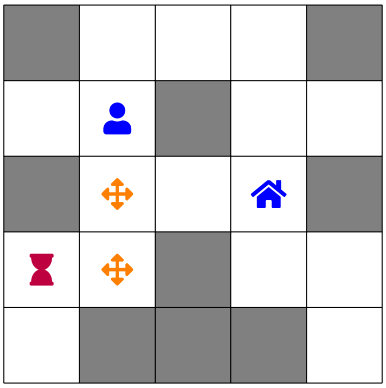
Rysunek 28: Oznacz "x":1,"y":2 jako kandydata



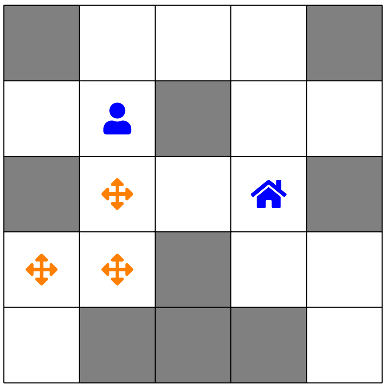
Rysunek 28: Wybierz "x":1,"y":1 jako następnie rozpatywany węzeł



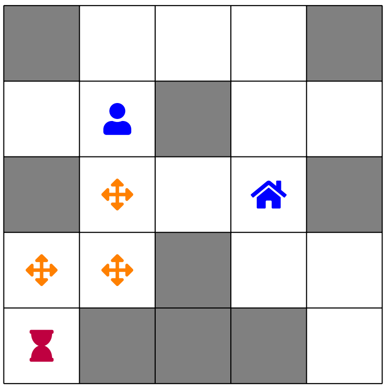
Rysunek 28: Oznacz "x":1,"y":1 jako kandydata



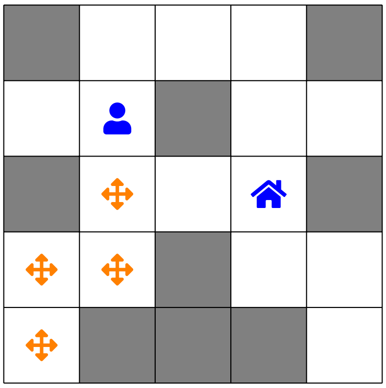
Rysunek 28: Wybierz "x":0,"y":1 jako następnie rozpatywany węzeł



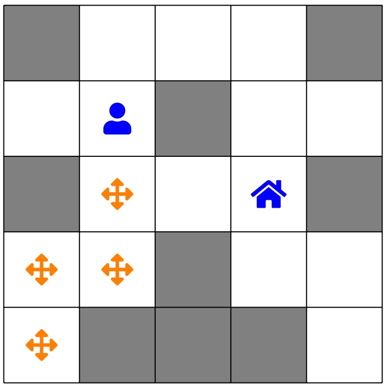
Rysunek 28: Oznacz "x":0,"y":1 jako kandydata



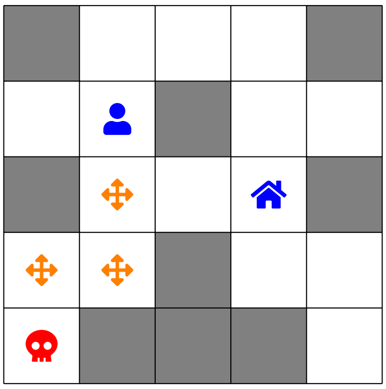
Rysunek 28: Wybierz "x":0,"y":0 jako następnie rozpatywany węzeł



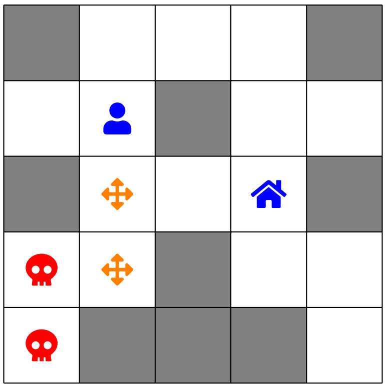
Rysunek 28: Oznacz "x":0,"y":0 jako kandydata



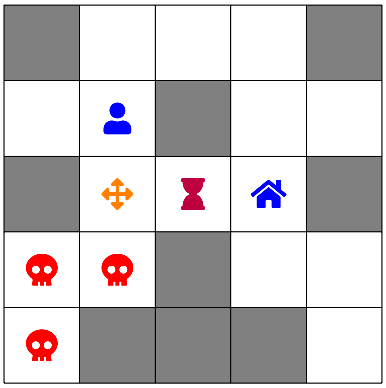
Rysunek 28: Oznacz "x":0,"y":0 jako zapomniany



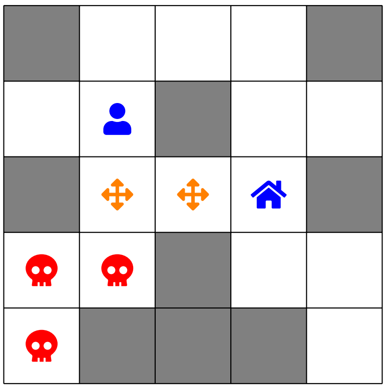
Rysunek 28: Oznacz "x":0,"y":1 jako zapomniany



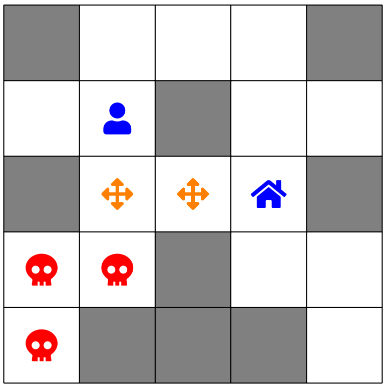
Rysunek 28: Oznacz "x":1,"y":1 jako zapomniany



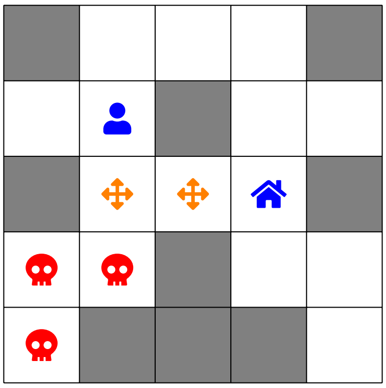
Rysunek 28: Wybierz "x":2,"y":2 jako następnie rozpatywany węzeł



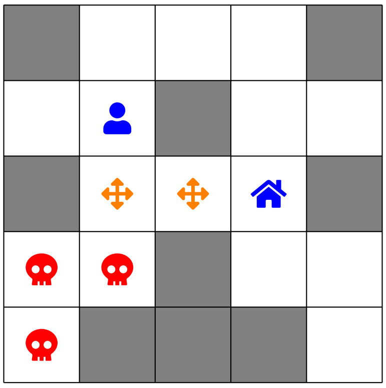
Rysunek 28: Oznacz "x":2,"y":2 jako kandydata



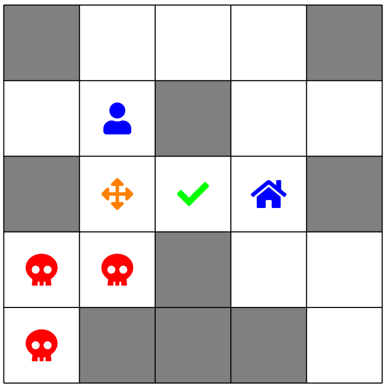
Rysunek 28: Wybierz "x":3,"y":2 jako następnie rozpatywany węzeł



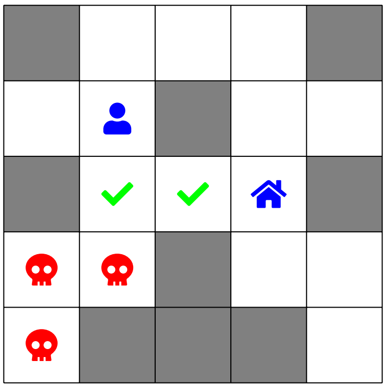
Rysunek 28: Oznacz "x":3,"y":2 jako kandydata



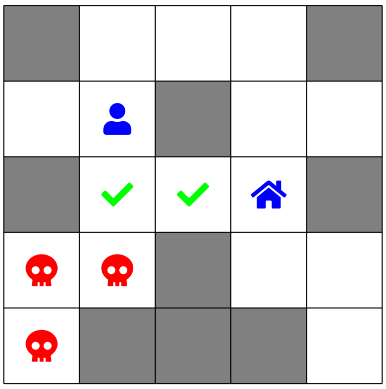
Rysunek 28: Wybierz "x":3,"y":2 do finalnej ścierzki



Rysunek 28: Wybierz "x":2,"y":2 do finalnej ścierzki



Rysunek 28: Wybierz "x":1,"y":2 do finalnej ścierzki



Rysunek 28: Wybierz "x":1,"y":3 do finalnej ścierzki

3.2 Algorytm BFS

3.2.1 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:

3.2.2 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)

3.2.3 Proces przeszukiwania BFS

Główna logika zaimplementowana jest w pętli przetwarzającej kolejkę:

Algorytm 2 Procedura BFS

Dopóki kolejka nie jest pusta **wykonaj**
  $current \leftarrow queue.dequeue()$
  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**
      $state[neighbour] \leftarrow current$ (zapisz pochodzenie)
      $queue.enqueue(neighbour)$
      Oznacz *neighbour* jako `queued`
    koniec jeśli
  koniec dla
koniec dopóki

3.2.4 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

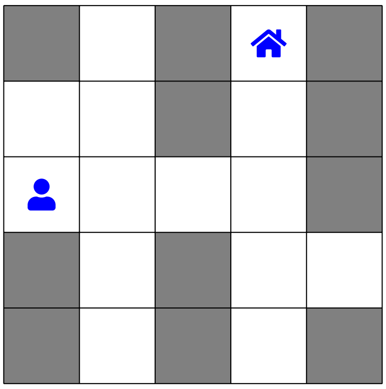
3.2.5 Złożoność obliczeniowa

- **Czasowa**: $O(V + E)$
  - 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

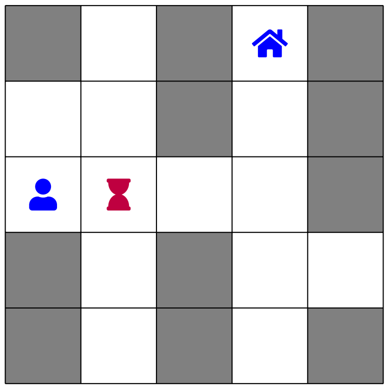
3.2.6 Właściwości algorytmu

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

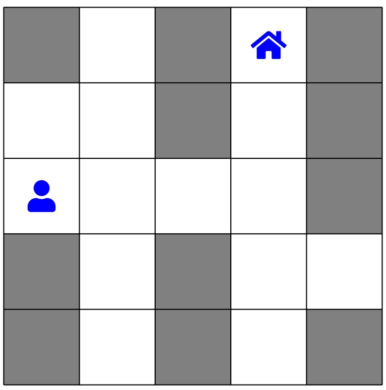
Przykład działania algorytmu przedstawia rysunek 29.



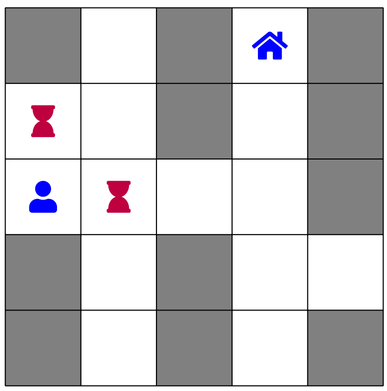
Rysunek 29: Dodaj do kolejki węzeł x: 0 y: 2



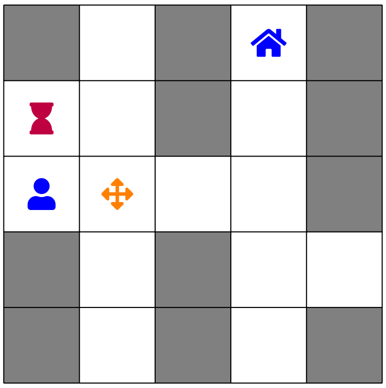
Rysunek 29: Dodaj do kolejki węzeł x: 1 y: 2



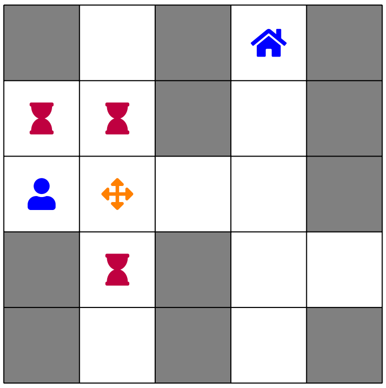
Rysunek 29: Rozpatrz x: 0 y: 2



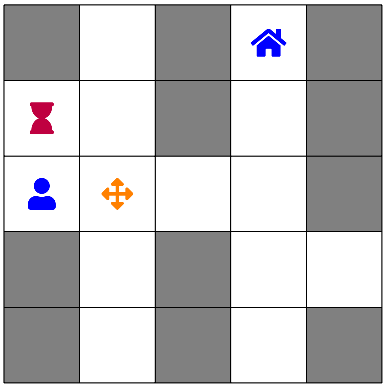
Rysunek 29: Dodaj do kolejki węzeł x: 0 y: 3



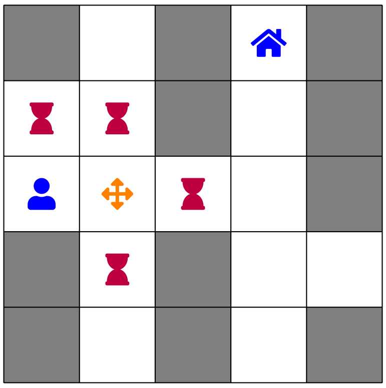
Rysunek 29: Rozpatrz x: 1 y: 2



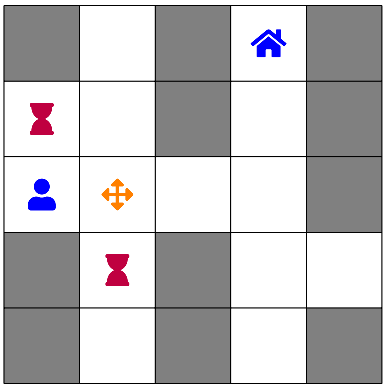
Rysunek 29: Dodaj do kolejki węzeł x: 1 y: 3



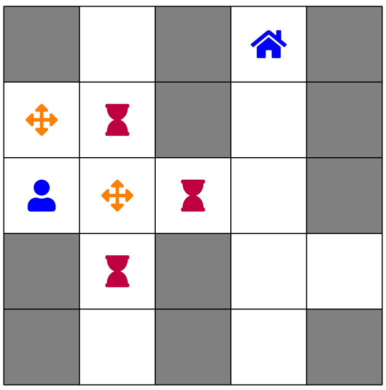
Rysunek 29: Dodaj do kolejki węzeł x: 0 y: 2



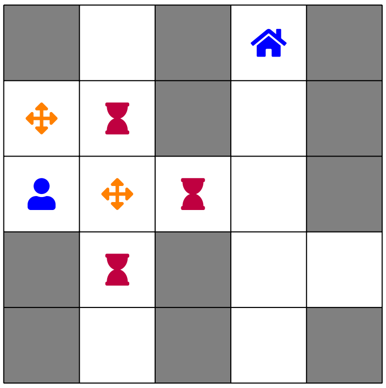
Rysunek 29: Dodaj do kolejki węzeł x: 2 y: 2



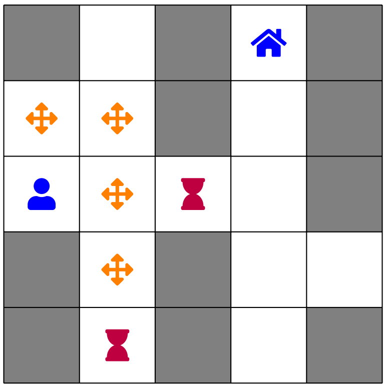
Rysunek 29: Dodaj do kolejki węzeł x: 1 y: 1



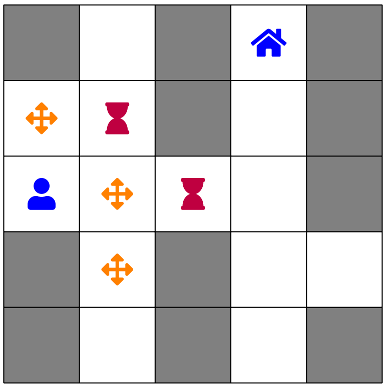
Rysunek 29: Rozpatrz x: 0 y: 3



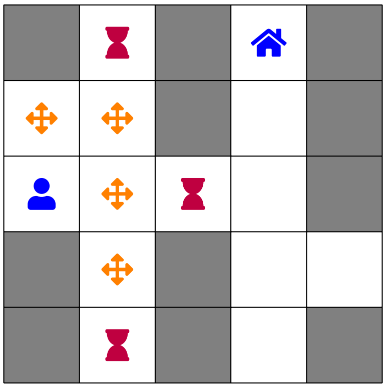
Rysunek 29: Rozpatrz x: 0 y: 2



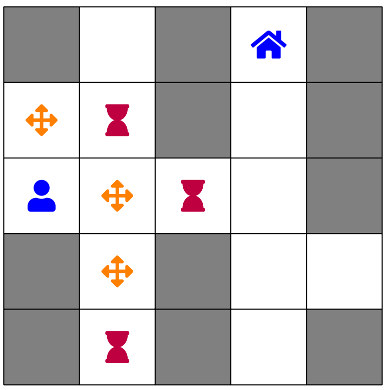
Rysunek 29: Rozpatrz x: 1 y: 3



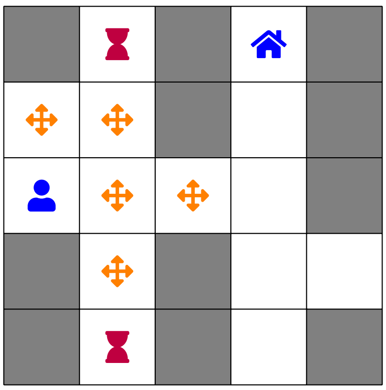
Rysunek 29: Rozpatrz x: 1 y: 1



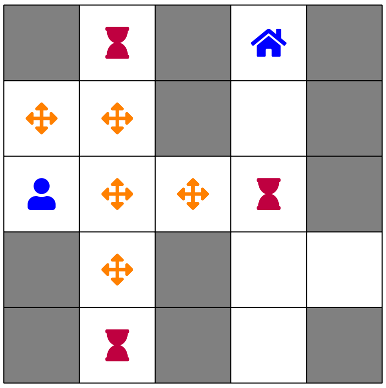
Rysunek 29: Dodaj do kolejki węzeł x: 1 y: 4



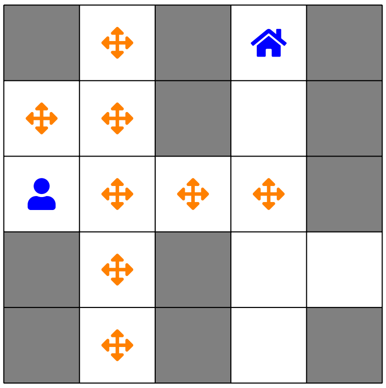
Rysunek 29: Dodaj do kolejki węzeł x: 1 y: 0



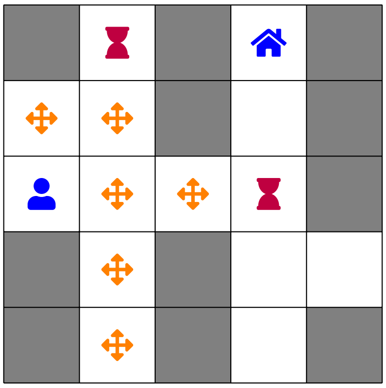
Rysunek 29: Rozpatrz x: 2 y: 2



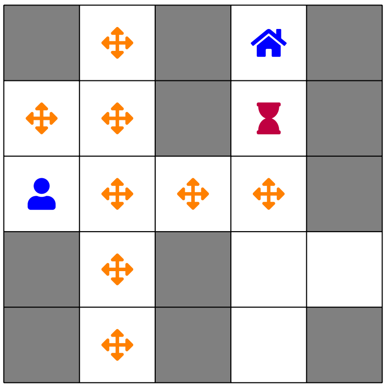
Rysunek 29: Dodaj do kolejki węzeł x: 3 y: 2



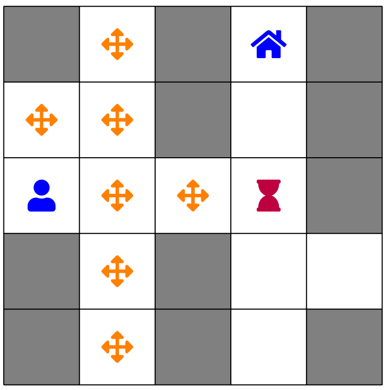
Rysunek 29: Rozpatrz x: 3 y: 2



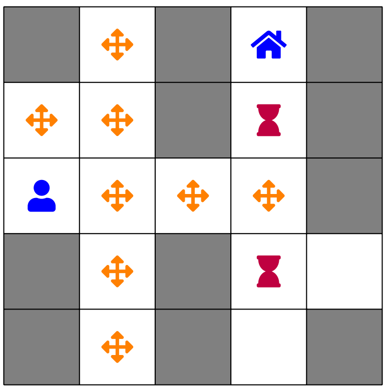
Rysunek 29: Rozpatrz x: 1 y: 0



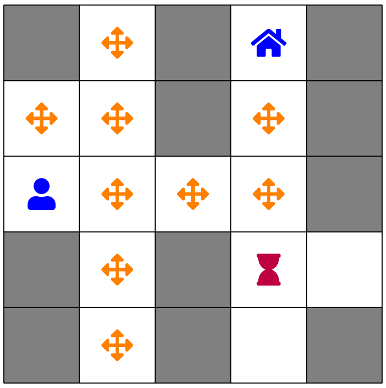
Rysunek 29: Dodaj do kolejki węzeł x: 3 y: 3



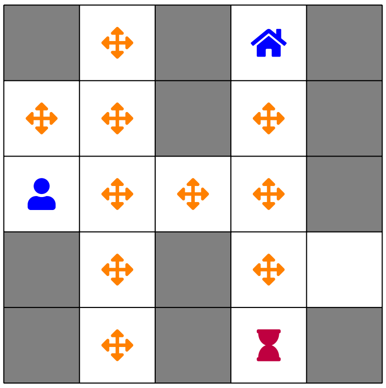
Rysunek 29: Rozpatrz x: 1 y: 4



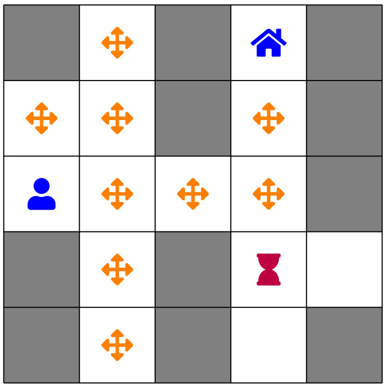
Rysunek 29: Dodaj do kolejki węzeł x: 3 y: 1



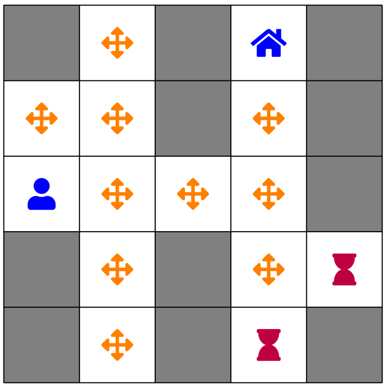
Rysunek 29: Rozpatrz x: 3 y: 3



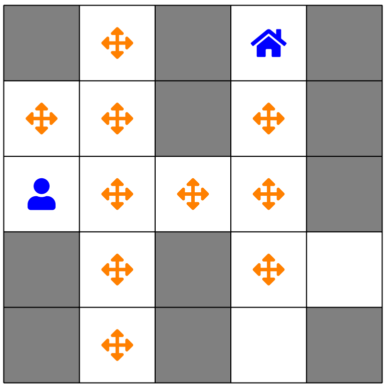
Rysunek 29: Dodaj do kolejki węzeł x: 3 y: 0



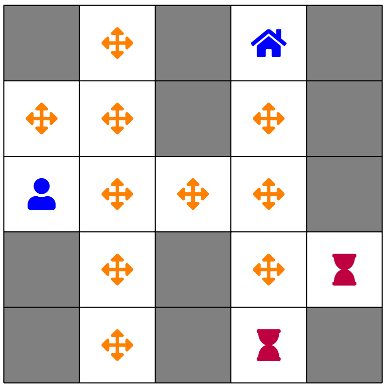
Rysunek 29: Dodaj do kolejki węzeł x: 3 y: 4



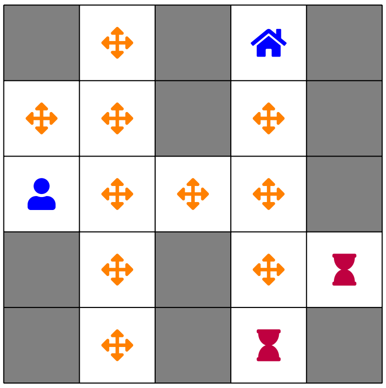
Rysunek 29: Dodaj do kolejki węzeł x: 4 y: 1



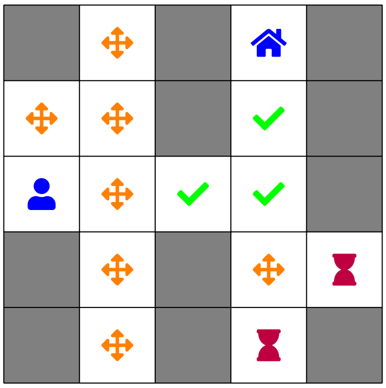
Rysunek 29: Rozpatrz x: 3 y: 1



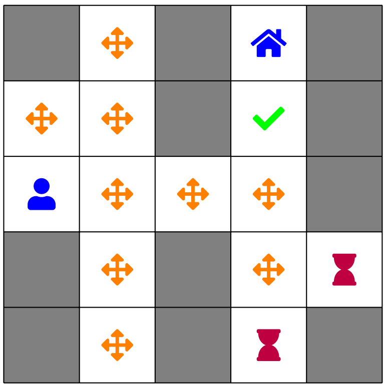
Rysunek 29: Rozpatrz x: 3 y: 4



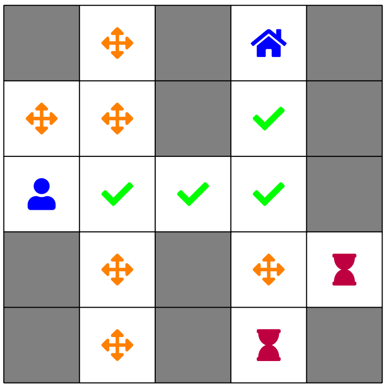
Rysunek 29: Wybierz x: 3 y: 4 do finalnej ścierzki



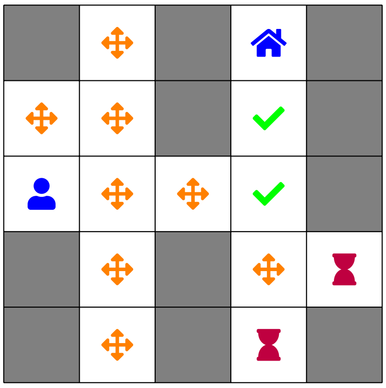
Rysunek 29: Wybierz x: 2 y: 2 do finalnej ścierzki



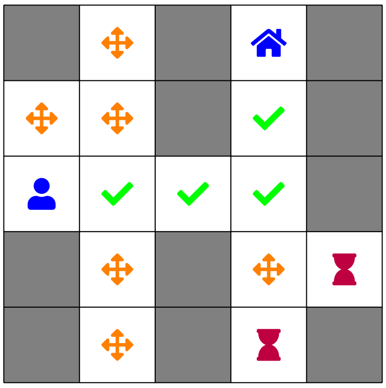
Rysunek 29: Wybierz x: 3 y: 3 do finalnej ścierzki



Rysunek 29: Wybierz x: 1 y: 2 do finalnej ścierzki



Rysunek 29: Wybierz x: 3 y: 2 do finalnej ścierzki



Rysunek 29: Wybierz x: 0 y: 2 do finalnej ścierzki

3.3 Algorytm A*

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 wszerz (BFS) i zachłannego przeszukiwania najlepszego pierwszego (Best-First Search). Działa poprzez minimalizację funkcji kosztu:

$$f(n) = g(n) + h(n)$$

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)**:

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

Zapewnia dopuszczalność (nie przeszacowuje kosztu).

3.4.4 Główna pętla algorytmu

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

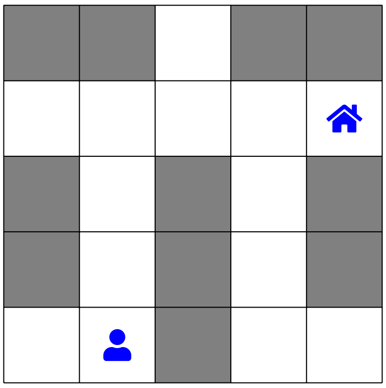
3.4.5 Budowanie ścieżki

1. **Śledzenie wsteczne:**
   - Rozpocznij od mety
   - Podążaj do poprzedników aż do startu
   - Oznaczaj węzły jako `selected`
2. **Końcowe przetwarzanie:**
   - Odwróć ścieżkę (start → meta)
   - Oznacz start jako `selected`

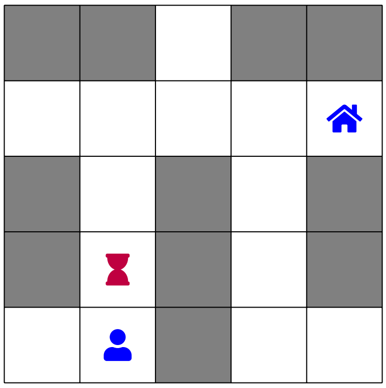
3.4.6 Złożoność obliczeniowa

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

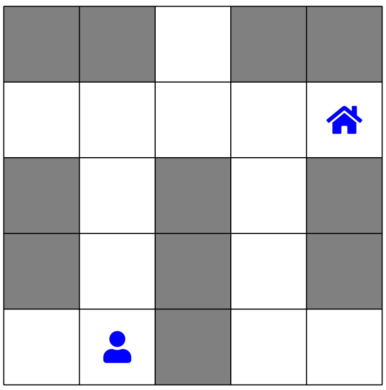
Przykład działania algorytmu przedstawia rysunek 30.



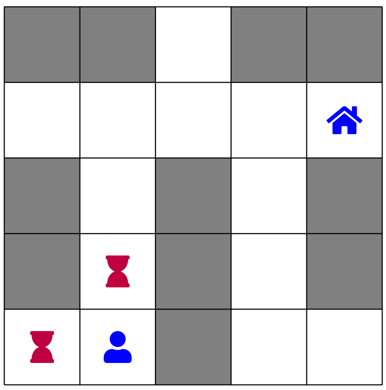
Rysunek 30: Dodaj do kolejki węzeł x: 1 y: 0



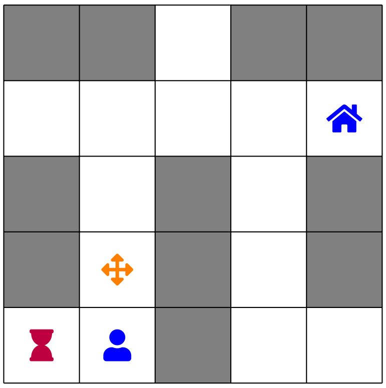
Rysunek 30: Dodaj do kolejki węzeł x: 1 y: 1



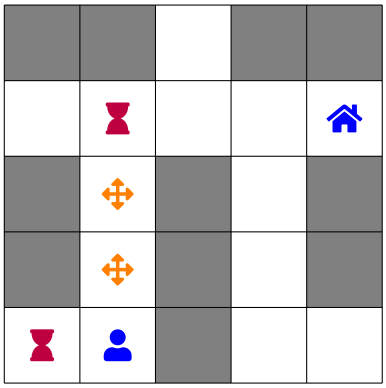
Rysunek 30: Rozpatrz x: 1 y: 0



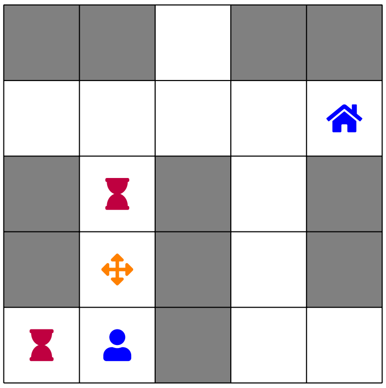
Rysunek 30: Dodaj do kolejki węzeł x: 0 y: 0



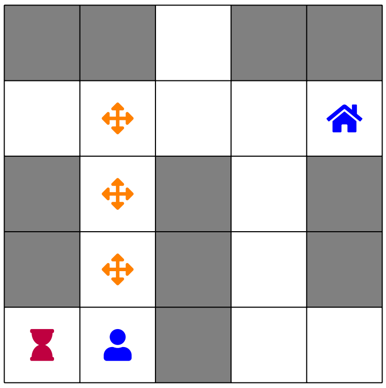
Rysunek 30: Rozpatrz x: 1 y: 1



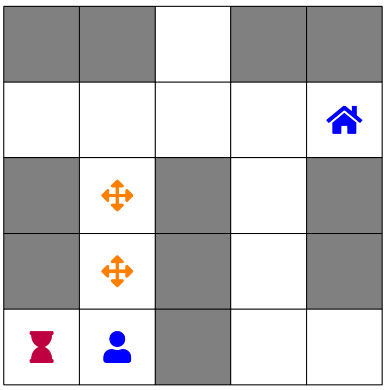
Rysunek 30: Dodaj do kolejki węzeł x: 1 y: 3



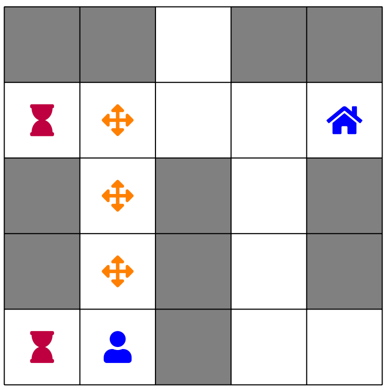
Rysunek 30: Dodaj do kolejki węzeł x: 1 y: 2



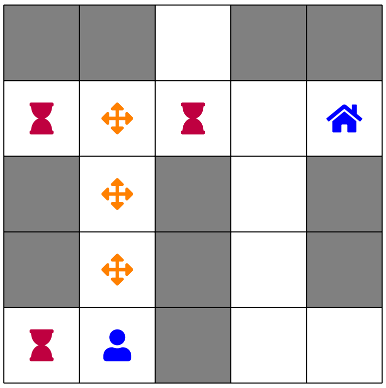
Rysunek 30: Rozpatrz x: 1 y: 3



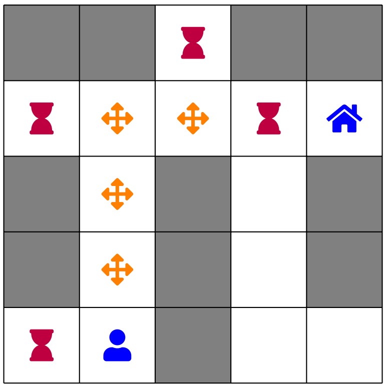
Rysunek 30: Rozpatrz x: 1 y: 2



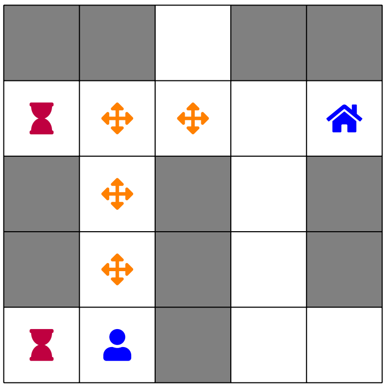
Rysunek 30: Dodaj do kolejki węzeł x: 0 y: 3



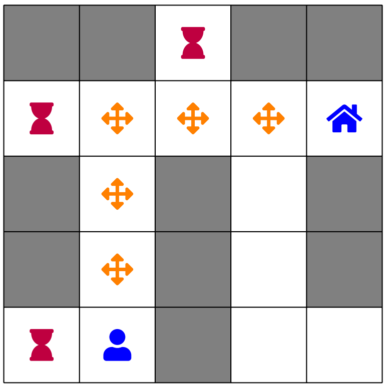
Rysunek 30: Dodaj do kolejki węzeł x: 2 y: 3



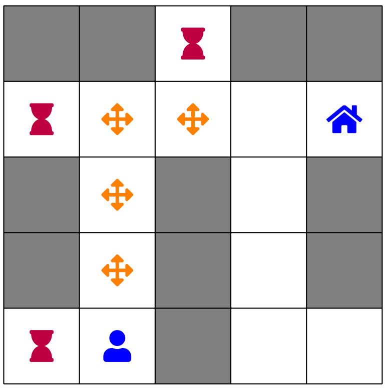
Rysunek 30: Dodaj do kolejki węzeł x: 3 y: 3



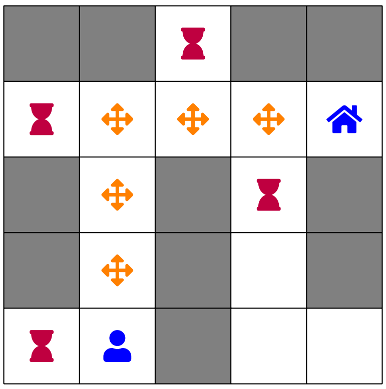
Rysunek 30: Rozpatrz x: 2 y: 3



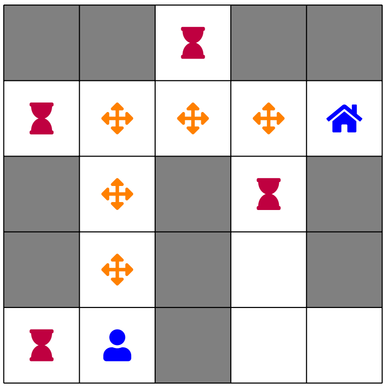
Rysunek 30: Rozpatrz x: 3 y: 3



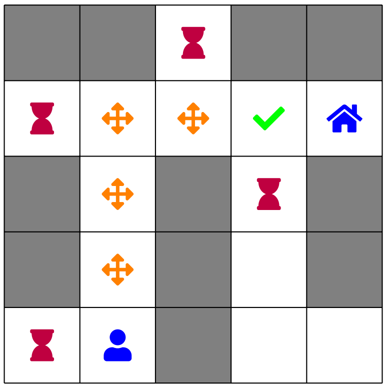
Rysunek 30: Dodaj do kolejki węzeł x: 2 y: 4



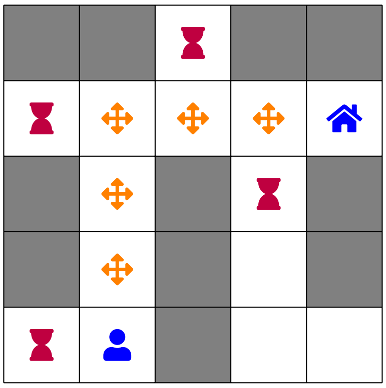
Rysunek 30: Dodaj do kolejki węzeł x: 3 y: 2



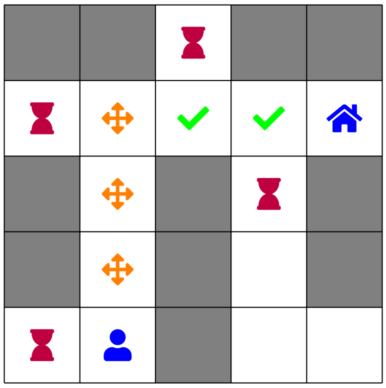
Rysunek 30: Dodaj do kolejki węzeł x: 4 y: 3



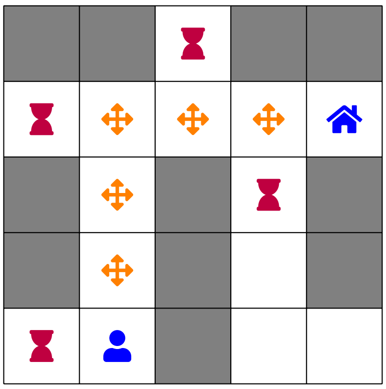
Rysunek 30: Wybierz x: 3 y: 3 do finalnej ścierzki



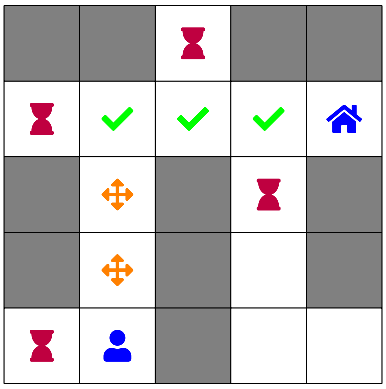
Rysunek 30: Rozpatrz x: 4 y: 3



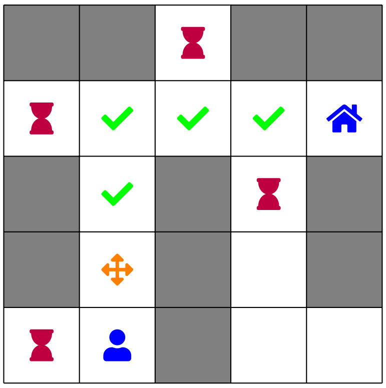
Rysunek 30: Wybierz x: 2 y: 3 do finalnej ścierzki



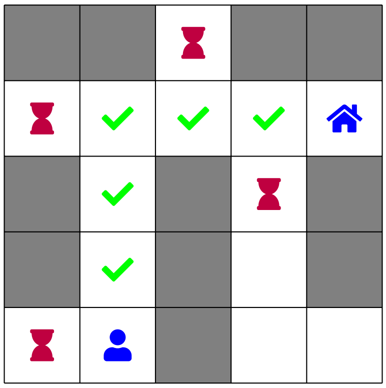
Rysunek 30: Wybierz x: 4 y: 3 do finalnej ścierzki



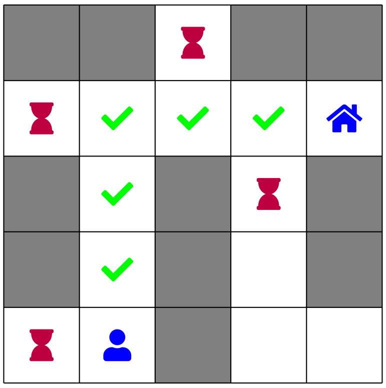
Rysunek 30: Wybierz x: 1 y: 3 do finalnej ścierzki



Rysunek 30: Wybierz x: 1 y: 2 do finalnej ścierzki



Rysunek 30: Wybierz x: 1 y: 0 do finalnej ścierzki



Rysunek 30: Wybierz x: 1 y: 1 do finalnej ścierzki

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.