Założenia - Labirynt i algorytm A*

1. Ściany labiryntu są reprezentowane jako przestrzenie między polami.

2. Labirynt reprezentowany za pomocą grafu. W węźle przechowywana informacja o położeniu za pomocą koordynatów X i Y, oraz o sąsiedztwie za pomocą listy wskaźników na sąsiadów.

3. Z danego pola można się poruszyć tylko w 4 kierunkach (lewo, prawo, góra, dół).

Te trzy założenia są ze sobą dosyć ściśle powiązane. Wszystko, co ich dotyczy, znajduje się w klasie Node (plik Node.h).
Jeśli chodzi o sąsiedztwo i ściany, jest to opisane dosłownie jedną linijką:

```
Node *next[4];
```

Jeśli węzeł ma dostęp sąsiada z danej strony (enum Side jest używany do wyboru kierunku), pojawi się tam adres następnego węzła, a w wypadku ściany – nullptr. Cztery kierunki są również zrealizowane przez zastosowanie czterech pól w tablicy i enuma. Dzięki połączeniu za pomocą wskaźników, można bezproblemowo przechodzić z punktu do punktu, po grafie, z czego też korzysta algorytm A*.

Koordynaty X i Y węzła są dostępne w postaci pól w klasie:

```
int x;
int y;
```

4. Z każdego punktu w labiryncie da się dojść do wyjścia.

Do wygenerowania labiryntu użyty został iteracyjny algorytm Depth-First Search – funkcja Maze::generateMaze() w pliku Maze.cpp:75. Przechodzi on przez wszystkie węzły labiryntu, dzięki czemu z każdego pola można dojść do dowolnego innego, jedną ścieżką.

5. Funkcja heurystyczna obliczana jako odległość punktu w labiryncie od punktu wyjścia za pomocą równania: |X.wyjścia - X.dane| + |Y.wyjścia – Y.dane|.

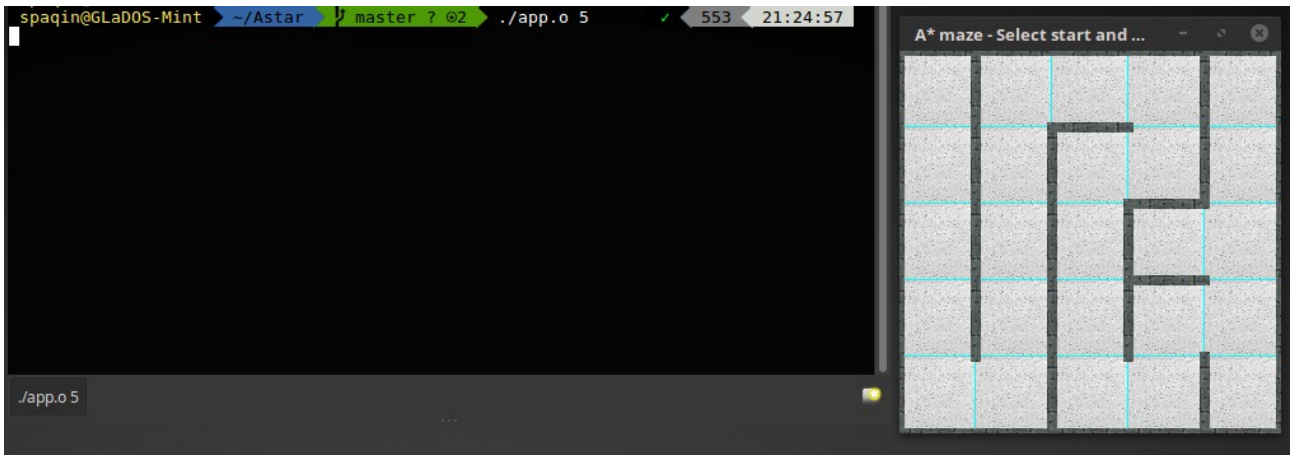
W funkcji step() w pliku Astar.cpp:37, obliczana jest wartość dla nowego, nieodkrytego węzła tym kawałkiem kodu:

```
currentNode->next[i]->setPriority((
     abs(currentNode->next[i]->getX()- goalNode->getX()) +
     abs(currentNode->next[i]->getY() - goalNode->getY() )) +
     (currentNode->getPassedRoute() + 1));
```

Składa się on z części heurystycznej (takiej, jak w założeniu), obliczenia odległości przebytej już drogi i zsumowania obu wartości

6. Wymiary labiryntu zadane jako dynamiczny parametr n na początku działania programu.

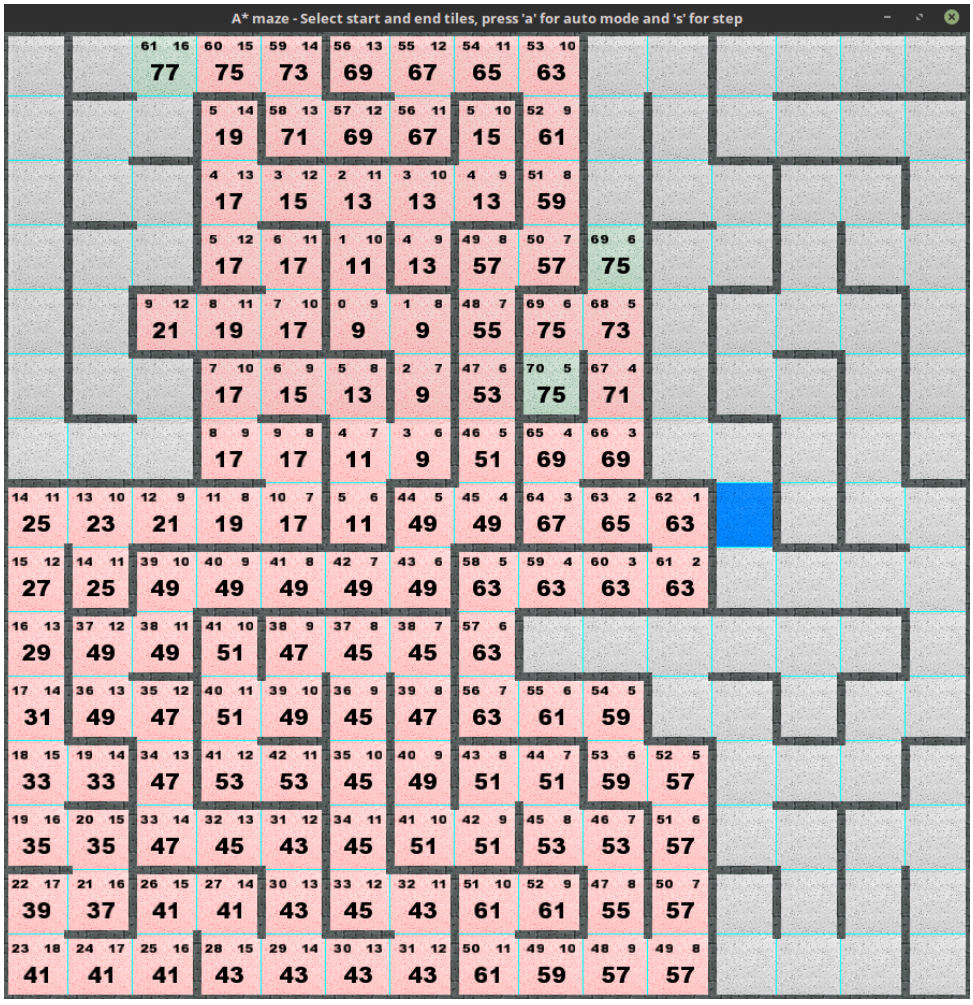
Main.cpp:27 – ustawienie rozmiaru labiryntu jako parametr przekazywany z konsoli przed uruchomieniem. Efekt:



Domyślnym parametrem jest n=15.

Założenia interfejs użytkownika:

1. Wizualizacja całego labiryntu, ścian, pól początku i końca oraz naniesienie na pola wartości funkcji heurystycznej i przebytej drogi w formie tekstowej



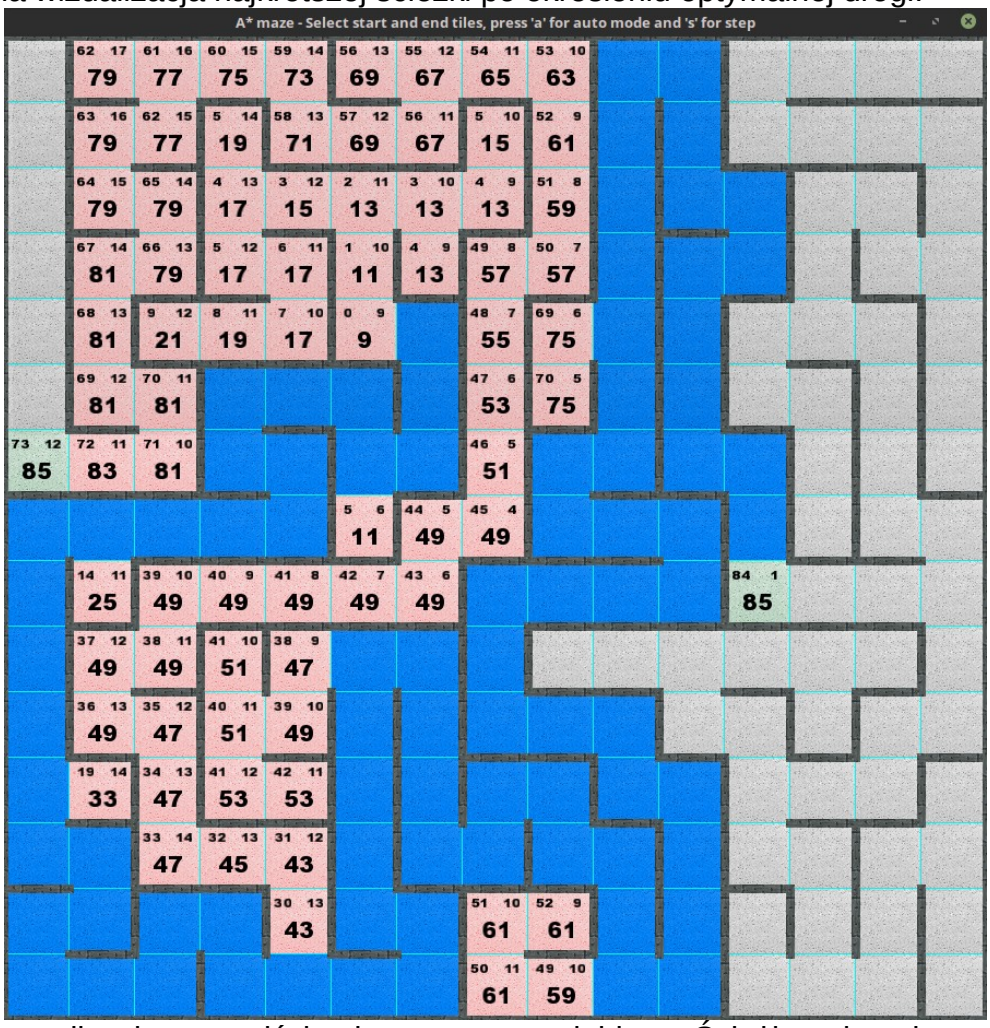
Oto okno przykładowego wywołania programu. Po wybraniu punktu początkowego i punktu końcowego oraz wykonaniu części kroków, rzuca się w oczy niebieski kolor płytki końcowej oraz cyferek na płytkach już “przemielonych” przez algorytm. Znaczenie liczb wygląda następująco:
- Lewy górny róg – odległość (faktyczna) węzła od węzła początkowego (węzeł początkowy ma wartość 0)
- Prawy górny róg – wartość funkcji heurystycznej (zgodnej z założeniami wyżej)
- Środek – suma.

Źródło: plik Display.cpp:187 (wpisanie danych), AlgorithmWrapper.cpp (kolorowanie)

2. Możliwość realizacji algorytmu krok po kroku w kolejności - wskazanie kolejnego elementu - przejście, wskazanie kolejnego elementu, przejście itp.

Po naciśnięciu przycisku “S” na klawiaturze, wykonywany jest kolejny krok algorytmu. Kolejne, rozpatrywane w tej chwili węzły zaznaczone są kolorem zielonym. Kolejne naciśnięcia tego przycisku spowodują dokładnie ten sam efekt. Naciśnięcie przycisku “A” spowoduje automatyczne wykonanie kolejnych kroków, wraz z wizualizacją.
Źródło: AlgorithmWrapper.cpp

3. Graficzna wizualizacja najkrótszej ścieżki po określeniu optymalnej drogi.



Oto okno po całkowitym przejściu algorytmu przez labirynt. Ścieżka od punktu początkowego, do końcowego zaznaczona jest niebieskimi kafelkami. Widać też węzły, które zostały przez algorytm sprawdzone.

Za odtworzenie i pokolorwanie ścieżki odpowiedzialna jest funkcja AlgorithmWrapper::drawPassedRoute() (AlgorithmWrapper.cpp:52)