**Tworzenie labiryntu i szukanie ścieżki wyjścia z niego**

Do rozwiązania tego problemu stworzyłem kilka głównych funkcji:

1) **draw\_grid**() – funkcja ta miała na celu stworzenie w oknie kratownicy, która stanowiła podstawę do stworzenia labiryntu. Tworzona kratownica posiadała zadaną liczbę kolumn i wierszy, które zbudowane były z komórek - elementów klasy **Cell**. Każda z nich początkowo posiadała 4 ściany – górną, dolną, prawą i lewą. W późniejszych krokach programu, odpowiednie ściany były usuwane, budując jednocześnie labirynt.   
Tworzone w tej funkcji komórki były dodawane do słownika **grid** jako jego klucze. Wartości dla nich stanowiły natomiast listy sąsiadujących z tą komórką innych komórek, które uzupełniane były w późniejszych funkcjach programu.

2) **draw\_maze**() - funkcja służąca do stworzenia labiryntu. Jej działanie opierało się na podejściu **recursive backtracking**. W myśl jego, stworzone zostały dwie listy: **stack** która działała jako stos do przechowywania komórek, po których ma przechodzić funkcja, oraz **visited**, która przechowywała już odwiedzone komórki. Głównym elementem funkcji była pętla, wykonująca się dopóki na stosie znajdowały się komórki. Na początku funkcja wybierała komórkę w lewym górnym rogu kratownicy, dodawała ją na stos i do listy komórek odwiedzonych. Następnie sprawdzane było, czy komórka posiada sąsiadów (komórki), którzy nie zostali jeszcze odwiedzeni. Jeżeli tak, odpowiedni kierunek był dodawany do listy **directions**. Następnie wśród nich wybierany był kierunek, oraz odpowiadająca mu komórka, która była wykorzystywana w kolejnej iteracji pętli. W słowniku była wtedy ustanawiana relacja sąsiedztwa między dwoma komórkami – była ona potrzebna przy późniejszym szukaniu ścieżki wyjścia z labiryntu. Między komórkami zostawała również usunięta ściana przed nadpisanie, były one również zaznaczane odpowiednim kolorem jako komórki tworzące labirynt.  
Jeśli do listy **directions** nie dodano żadnego kierunku, wybierana była komórka ze szczytu stosu, jednocześnie następowało jej usunięcie z tego stosu. Było to równoważne sytuacji, kiedy wokół danej komórki wszystkie inne zostały już odwiedzone (dodane do labiryntu), w związku z czym program cofał się do czasu, aż natrafi na komórkę, której choć jeden sąsiad nie był jeszcze odwiedzony.

3**) find\_path**() – służyła do znalezienia ścieżki wyjścia z labiryntu. W założeniu ścieżka miała mieć swój początek w lewym górnym rogu labiryntu, a zakończyć w prawym dolnym rogu. W funkcji tej zostało użyte podobne podejście jak w funkcji **draw\_maze**(). Listy **stack** i **visited** jak poprzednio reprezentowały stos oraz listę komórek już odwiedzonych. Głównym elementem funkcji była pętla while, która wykonywała się do momentu, aż komórka w której się znaleźliśmy nie była komórką końcową. Na początku do pętli program wchodził z komórką startową, którą dodawał do listy komórek odwiedzonych oraz na stos. Następnie do listy **neighbours** dodawani byli sąsiedzi komórki, do których możemy bezpośrednio przejść, a którzy nie zostali wcześniej odwiedzeni. Jeśli komórka takowych posiadała, wybierany był jeden z nich i od niego startował kolejny przebieg pętli. W przeciwnym przypadku, pobierana była komórka ze szczytu stosu, następnie od niej startowała kolejna iteracja.