2.2.5. Algorytm A*

Obieranie cebuli warstwa po warstwie, jak w algorytmie przeszukiwania wszerz, może być bardzo czasochłonne. Podobnie jak BFS, algorytm A* próbuje znaleźć najkrótszą drogę ze stanu początkowego do docelowego. W odróżnieniu od przedstawionej implementacji przeszukiwania BFS algorytm A* wykorzystuje kombinację funkcji kosztów i funkcji heurystycznej do koncentrowania się na drogach, które dają największą szansę na szybkie znalezienie rozwiązania.

Funkcja kosztu, g(n), oblicza koszt dojścia do określonego stanu. W przypadku naszego labiryntu byłaby nim liczba poprzednich kroków, przez jakie musieliśmy przejść, aby dotrzeć do danego stanu. Funkcja heurystyczna, h(n), zwraca szacowany koszt przejścia z danego stanu do stanu docelowego. Można udowodnić, że jeśli heurystyka h(n) jest dopuszczalna, odnaleziona droga będzie optymalna. Dopuszczalna funkcja heurystyczna nigdy nie przeszacowuje kosztu drogi do celu. Przykładem na płaszczyźnie dwuwymiarowej może być odległość w linii prostej, ponieważ linia prosta zawsze jest najkrótszą drogą[1].

Całkowity koszt dla dowolnego analizowanego stanu reprezentuje funkcja f(n), która jest prostą kombinacją funkcji g(n) i h(n). Faktycznie, f(n) = g(n) + h(n). Wybierając ze zbioru `frontier` następny stan do sprawdzenia, algorytm przeszukiwania A* wybiera ten o najmniejszej wartości f(n). To odróżnia go od algorytmów BFS oraz DFS.

KOLEJKI PRIORYTETOWE

Aby ułatwiać wybieranie ze zbioru `frontier` stanu o najniższej wartości f(n), algorytm A* opiera ten zbiór na strukturze danych *kolejka priorytetowa*. Kolejka priorytetowa przechowuje elementy w określonej kolejności, aby pierwszy ściągany z niej element zawsze miał najwyższy priorytet (w naszym przypadku elementem o najwyższym priorytecie jest ten o najniższej wartości f(n)). Zazwyczaj wiąże się to z wykorzystaniem w roli pojemnika sterty binarnej, co oznacza operacje *push* o koszcie O(lg n) i *pop* o koszcie O(lg n).

Biblioteka standardowa Pythona oferuje funkcje `heappush()` i `heappop()`, które przechowują otrzymaną listę w postaci sterty binarnej. Możemy zaimplementować kolejkę priorytetową, budując cienką otokę wokół tych funkcji biblioteki standardowej. Nasza klasa `PriorityQueue` będzie przypominała klasy stosu typu `Stack` i kolejki typu `Queue`, z metodami `push()` i `pop()` zmodyfikowanymi w taki sposób, aby bazowały one na metodach `heappush()` oraz `heappop()`.

Listing 2.25. generic_search.py kontynuacja

```
class PriorityQueue(Generic[T]):
    def __init__(self) -> None:
        self._container: List[T] = []

    @property
    def empty(self) -> bool:
        return not self._container # not zwraca true dla pustego pojemnika
```

[1] Dodatkowe informacje o heurystykach można znaleźć w książce Stuarta Russella i Petera Norviga zatytułowanej *Artificial Intelligence: A Modern Approach*. Wyd. 3. Pearson, 2010, s. 94.

```
    def push(self, item: T) -> None:
        heappush(self._container, item) # dodawanie według priorytetu

    def pop(self) -> T:
        return heappop(self._container) # ściąganie według priorytetu

    def __repr__(self) -> str:
        return repr(self._container)
```

Do porównywania priorytetów różnych elementów metody `heappush()` i `heappop()` używają operatora `<`. Dlatego we wcześniejszej części rozdziału musieliśmy zaimplementować metodę `__lt__()` w klasie `Node`. Porównując ze sobą dwa obiekty `Node`, bazujemy na ich wartościach f(*n*), które są po prostu sumą właściwości `cost` i `heuristic`.

HEURYSTYKI

Heurystyka określa intuicyjny sposób rozwiązywania problemu[2]. W kontekście rozwiązywania labiryntów heurystyki dążą do wybrania najlepszego kolejnego miejsca do przeszukania, które doprowadzi nas do celu. Innymi słowy, służą do domyślania się, które węzły ze zbioru frontier znajdują się najbliżej celu. Jak już wspomnieliśmy, jeśli heurystyka użyta w algorytmie przeszukiwania A* zwraca trafny względny wynik i jest dopuszczalna (nigdy nie przeszacowuje odległości), algorytm A* znajdzie najkrótszą ścieżkę. Heurystyki, które zwracają mniejsze wartości, prowadzą do przeszukiwania większej liczby stanów, natomiast heurystyki bliższe prawdziwej rzeczywistej odległości (ale jej nieprzekraczające, ponieważ wtedy byłyby niedopuszczalne) prowadzą do przeszukania mniejszej liczby stanów. W związku z tym idealne heurystyki maksymalnie zbliżają się do rzeczywistej odległości, nigdy jej nie przekraczając.

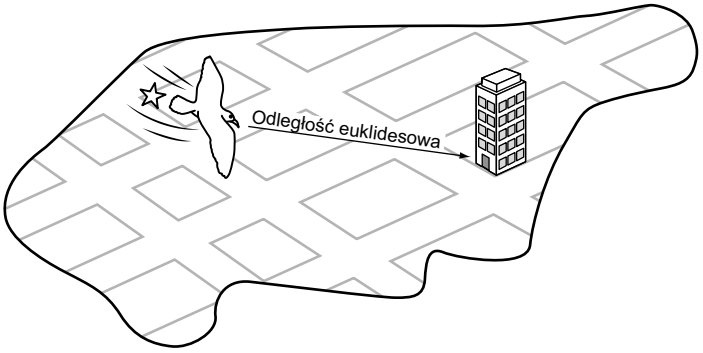
ODLEGŁOŚĆ EUKLIDESOWA

Jak się nauczyliśmy na lekcjach matematyki, najkrótsza droga między dwoma punktami to linia prosta. W związku z tym heurystyka oparta na prostej linii zawsze będzie dopuszczalnym rozwiązaniem problemu labiryntu. Do obliczania odległości euklidesowej służy następujący wzór, wyprowadzony z twierdzenia Pitagorasa: `odległość = √((różnica pozycji x)² + (różnica pozycji y)²)`. W przypadku naszych labiryntów różnica *x* stanowi odpowiednik różnicy numerów kolumn między pozycjami w labiryncie, natomiast różnica *y* stanowi odpowiednik różnicy numerów wierszy. To rozwiązanie implementujemy w pliku maze.py.

Listing 2.26. maze.py kontynuacja

```
def euclidean_distance(goal: MazeLocation) -> Callable[[MazeLocation], float]:
    def distance(ml: MazeLocation) -> float:
        xdist: int = ml.column - goal.column
        ydist: int = ml.row - goal.row
        return sqrt((xdist * xdist) + (ydist * ydist))
    return distance
```

[2] Dodatkowe informacje o heurystyce A* do szukania ścieżek można znaleźć w rozdziale *Heuristics* w dokumencie Amita Patel zatytułowanym *Amit's Thoughts on Pathfinding*, http://mng.bz/z7O4.



Rysunek 2.6. Odległość euklidesowa jest równa długości prostej linii od początku do celu

`euclidean_distance()` to funkcja, która zwraca inną funkcję. Ten ciekawy wzorzec jest możliwy w Pythonie i innych językach, które traktują funkcje jak typ pierwszoklasowy. Funkcja `distance()` przechwytuje `goal`, czyli pozycję celu przekazywaną do funkcji `euclidean_distance()`. Przechwytywanie oznacza, że funkcja `distance()` może się odwoływać do tej zmiennej za każdym razem, gdy zostaje wywołana (na stałe). Zwracana funkcja wykorzystuje `goal` do przeprowadzenia własnych obliczeń. Ten wzorzec umożliwia tworzenie funkcji, która wymaga mniejszej liczby parametrów. Zwracana funkcja `distance()` ma tylko jeden parametr określający początkową pozycję w labiryncie i stale „zna" cel.

Na rysunku 2.6 zilustrowano odległość euklidesową w kontekście siatki, takiej jak ulice Manhattanu.

ODLEGŁOŚĆ MANHATTAŃSKA

Odległość euklidesowa to przydatne narzędzie, ale do naszego konkretnego zadania (labiryntu, w którym można się poruszać tylko w jednym z czterech kierunków) możemy użyć czegoś jeszcze lepszego. Nazwa odległości manhattańskiej wywodzi się ze sposobu nawigowania po Manhattanie, najsłynniejszej dzielnicy Nowego Jorku, która jest zbudowana na planie siatki. Aby dostać się z dowolnego miejsca w inne dowolne miejsce na Manhattanie, trzeba przejść określoną liczbę przecznic w poziomie i określoną liczbę przecznic w pionie (na Manhattanie prawie nie ma ulic biegnących po ukośnej). Aby wyznaczyć odległość manhattańską, wystarczy znaleźć różnicę w wierszach między dwoma pozycjami w labiryncie i dodać do niej różnicę w kolumnach. Na rysunku 2.7 pokazano odległość manhattańską.

Listing 2.27. maze.py kontynuacja

```
def manhattan_distance(goal: MazeLocation) -> Callable[[MazeLocation], float]:
    def distance(ml: MazeLocation) -> float:
        xdist: int = abs(ml.column - goal.column)
        ydist: int = abs(ml.row - goal.row)
        return (xdist + ydist)
    return distance
```