Algorytm A*

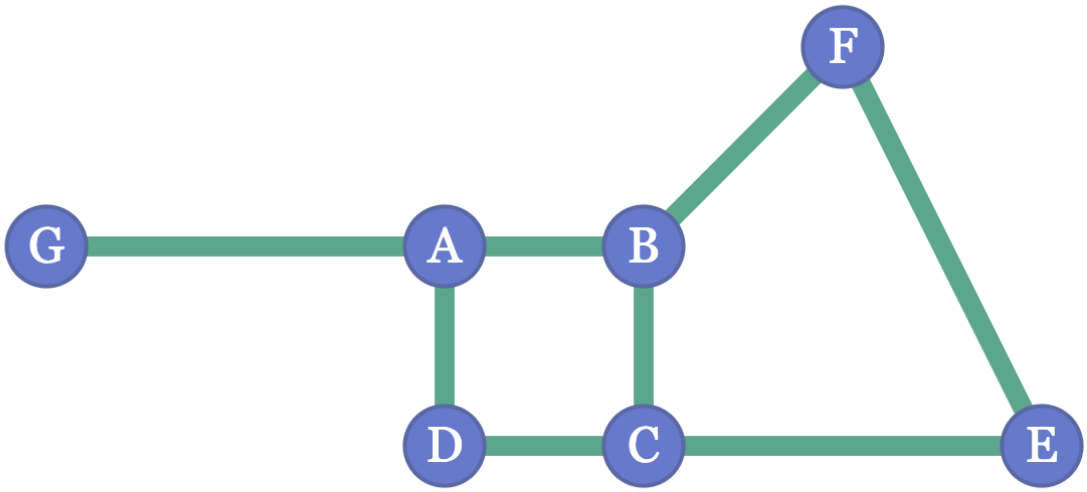
Karol Janik
Wydział Inżynierii Materiałowej i Fizyki
Politechnika Krakowska

23 czerwca 2020

1 Krótkie wprowadzenie do grafów

Grafy są nielinową strukturą danych, składających się z wezłów (ang. nodes) oraz krawędzi (ang. edges). Węzły (zwane również wierzchołkami) połączone są krawędziami. Dla każdego grafu możemy zdefiniować:

1. Zbiór węzłów
2. Zbiór wierzchołków (krawędzi)



Rysunek 1: Schemat grafu

Powyższy graf składa się:

1. Zbiór węzłów: **A B C D F G**.
2. Zbiór wierzchołkow dla każdego węzła:
   - **A**: A → B A → D A → G
   - **B**: B → A B → C B → F
   - **C**: C → B C → D C → E
   - **D**: D → C D → A
   - **E**: E → C E → F
   - **F**: F → B F → E
   - **G**: G → A

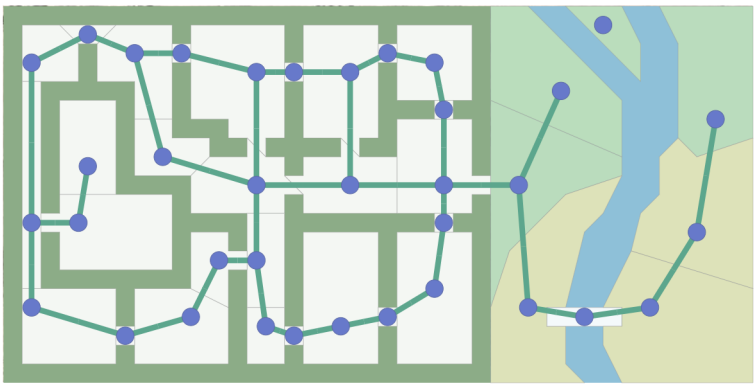
Grafy służą m.in do reprezentowania struktur miejskiej sieci telefonicznej lub wodociągowej. Znajdują one rownież zastosowanie wewnątrz portali społecznościowych. Dla przykładu, na facebooku każda osoba jest reprezentowana przez wierzchołek, a każdy węzeł jest strukturą i zawiera informacje takie jak ID osoby, imie, płeć itd..

2 Wprowadzenie do Algorytmu A*

W grach komputerowych, często chcemy znaleźć drogę pomiędzy dwoma punktami. Podczas szukania tej drogi niezawsze zależy nam na najkrótszej ścieżce, ale rownież ważnym aspektem jest czas podróży. Aby znaleźć tę drogę, możemy użyć algorytmu wyszukiwania opartego na grafach. A* jest popularnym algorytmem z tej grupy, jednak swoje

2.1 Reprezentacja mapy

Najważniejszą rzeczą, od której trzeba zacząć nauke algorytmów jest zrozumienie danych, które są na wejściu oraz na wyjściu.



Rysunek 2: Graf przekazywany na wejściu

Wejście: Algorytmy wyszukiwania bazujące na grafach, jak sama nazwa wskazuje, przyjmują graf na wejściu. Jest to zbiór lokalizacji (niebieskie kółka, zwane "węzłami") oraz połączeń (zielone linie, zwane "krawędziami") pomiędzy nimi. Algorytm nie widzi nic, prócz grafu. Nie widzi on tego, co coś jest w pomieszczeniu czy na zewnątrz lub jak duży obszar jest. Algorytm widzi tylko i wyłącznie graf!

Wyjście: Droga znaleziona przez algorytm A* słada się z węzłów i wierzchołków. A* każe Ci przejśćz jednego punktu do drugiego, ale nie wskaże Ci jak masz to zrobić. Cały czas musimy pamiętać, że algorytm ten nie wie nic na temat pomieszczeń lub drzwi, jedyne, co widzi to graf. Sam musisz zdecydować, co oznacza wierzchołek zwrócony przez algorytm, może to być przejście z płytki na płytkę, przejście w lini prostej, otwarcie drzwi, pływanie lub bieganie.

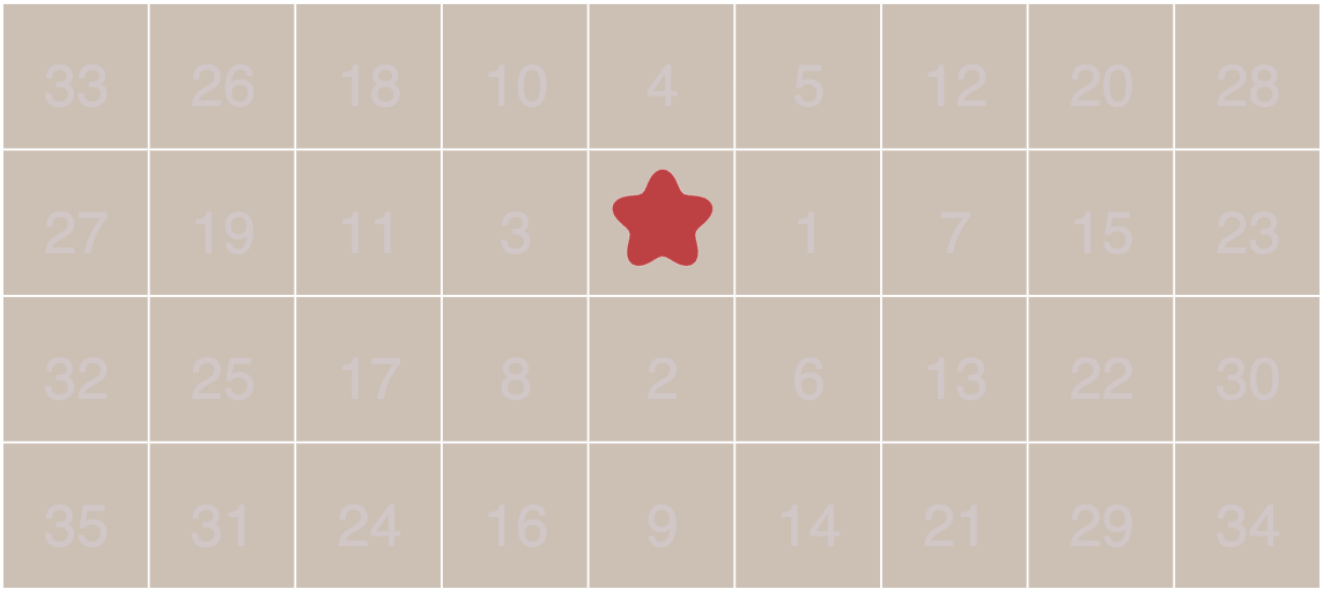
3 Przeszukwianie wszerz

Kluczową ideą wszystkich algorytmów, które oparte są na grafach jest śledzenie rozszerzających się pierścieni zwanych granicą (ang. frontier). W strukturach siatkowych proces ten jest czasami nazywany źalewaniem", ale ta sama technika działa w przypadku innych sieci niż siatki.

Jak zaimplementować ten algorytm? Wystarczy powtarzać poniższe kroki, dopóki granica będzie pusta:

1. Wybierz i usuń lokalizacje z granicy
2. Rozwiń go, patrząc na sąsiadów. Wszystkch sąsiadów, których jeszcze nie odwiedzilśmy, dodajemy do granicy, a także do zbioru lokalizacji odwiedzonych.

Zobaczmy to z bliska. Kafelki są ponumerowane w kolejności, w jakiej je odwiedzaliśmy: Imple-



Rysunek 3: Wizualizacja rozszerzających się pierścieni

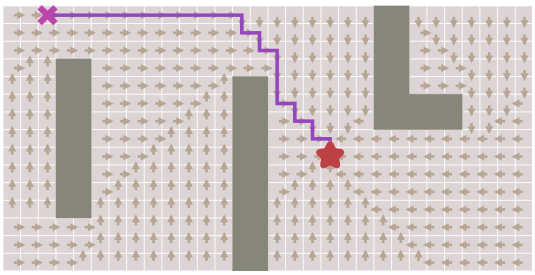
mentacja w pythonie:

```
frontier = Queue()
frontier.put(start)
visited = {}
visited[start] = True

while not frontier.empty():
    current = frontier.get()
    for next in graph.neighbors(current):
        if next not in visited:
        frontier.put(next)
        visited[next] = True
```

Powyższa pętla jest sercem algorytmów, które bazują na grafach. Ale jak za jej pomocą możemy znaleźć najkrótszą drogę? Pętla w zasadzie nie tworzy żadnej drogi, ona mówi tylko, jak odwiedzić wszystko na mapie. Wynika to z faktu, że przeszukiwanie wszerz ma dużo więcej zastosowań, niż szukanie drogi. Algorytm ten może być używany do tworzenia map odległościowych lub generwoania map proceduralnych oraz wielu innych ciekawych rzeczy. W tym artykule algoytm będzie wykorzystywany do wyszukiwania drogi, więc musimy zmodyfikować pętlę, aby śledzić, skąd przychodzimy dla każdej odwiedzonej lokalizacji i zmienić nazwę visited na cameFrom.

```
frontier = Queue()
frontier.put(start)
cameFrom = {}
cameFrom[start] = None

while not frontier.empty():
    current = frontier.get()
    for next in graph.neighbors(current):
        if next not in came_from:
        frontier.put(next)
        cameFrom[next] = current
```

Teraz cameFrom dla każdej lokalizacji wskazuje miejsce, z którego przybyliśmy.



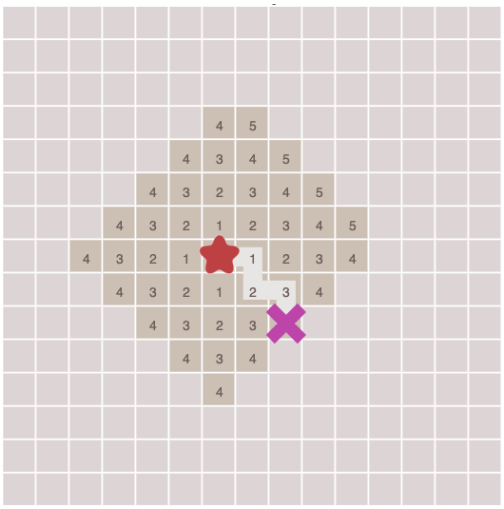
Kod do rekonstrukcji ścieżek jest prosty: podążaj za strzałkami wstecz od celu do początku. Scieżka jest sekwencją krawędzi, ale często łatwiej jest przechowywać węzły:

```
current = goal
path = []
while current != start:
    path.append(current)
    current = came_from[current]
path.append(start)
path.reverse()
```

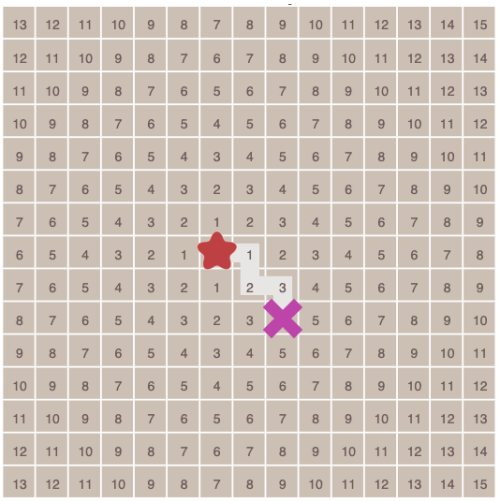
To najrpostszy algorytm wyszukwania ścieżki. Działa nie tylko na siatkach, jak pokazano tutaj, ale na dowolnej strukturze opartej o graf. W lochach lokalizacje grafów mogą być pokojami, a krawędzie drzwiami pomiędzy nimi. W platformówce krawędzie grafu mogą reprezentwoać działania, takie jak rcuh w lewo, ruch w prawo, skok w góre, skok w dół. Ogólnie, myśl o grafie jako o stanach i działaniach, które zmieniają stan.

3.1 Wczesne wyjście

Znaleźliśmy scieżki z jednej lokalizacji do wszystkich innych. Często zdarza się tak, że nie potrzebujemy wszystkich scieżek, potrzebujemy tylko ścieżki z jednek lokalizacji do drugiej. Możemy przestać rozszerzać granicę, gdy tylko osiągniemy nasz cel.



Rysunek 4: Z wczesnym wyjściem



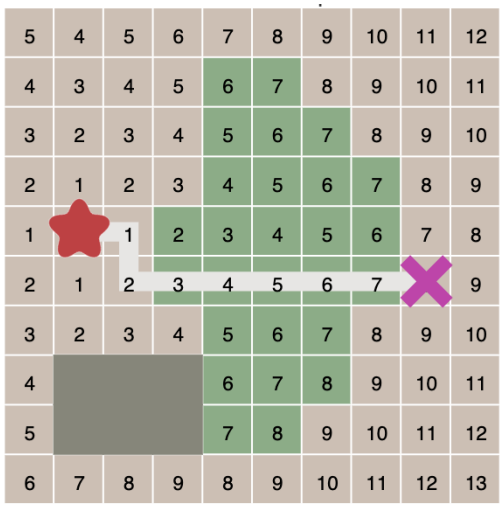
Rysunek 5: Bez wczesnego wyjścia

Kod jest prosty:

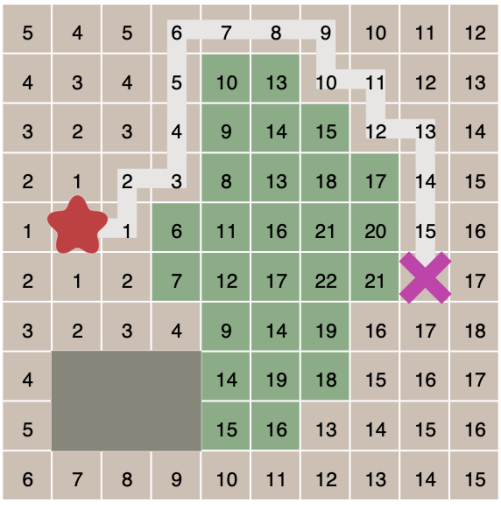
```
frontier = Queue()
frontier.put(start)
cameFrom = {}
cameFrom[start] = None

while not frontier.empty():
    current = frontier.get()

    if current == goal:
        break

    for next in graph.neighbors(current):
        if next not in cameFrom:
            frontier.put(next)
            cameFrom[next] = current
```

3.2 Koszty poruszania

Do tej pory poruszaliśmy się po drodzę, na której każdy krok "kosztował"tyle samo. W niektórych scenariuszach szukania drogi istnieją różnę "koszta"dla różnych ścieżek. Na przykład w grze Civilization, poruszanie się po płaskim terenie lub pustyni, kosztuje nas 1 punkt. Natomiast poruszanie sie po lesie lub wzgórzach kosztuje 5 punktów. Innym przykładem jest ruch po przekątnej na siatce, który kosztuje więcej, niż ruch osiowy. Chcemy, aby nasz algorytm uwzględnił te koszty. Porównajmy liczbę kroków od początku z odległością od początku:



Rysunek 6: Liczba kroków

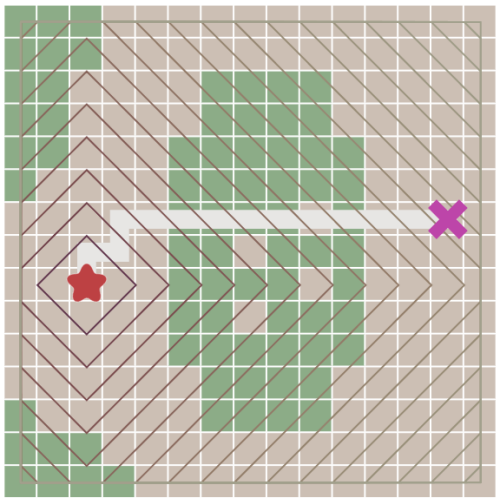


Rysunek 7: Dystans

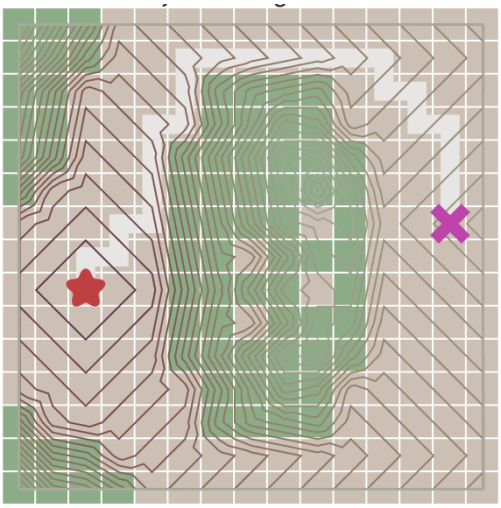
Do tego potrzebujemy użyc Algorytmu Dijkstry. Czym on się różni od wyszukiwania wszerz? Musimy śledzić koszty ruchu, więc wprowadźmy nową zmienną - costSoFar, która będzie przechowywać koszty poruszania się od początku do końca drogi. Podejmując decyzję o ocenie lokalizacji, chcemy wziąć pod uwagę koszty ruchu. Zatem, zamieńmy naszą kolejkę w kolejkę priorytetową. Mniej oczywiste jest to, że odwiedzimy lokalizację wiele razy, co wiąże się z różnymi kosztami, dlatego musimy nieco zmienić logikę. Zamiast dodawać lokalizację do granicy, jeżeli nie była ona nigdy odwiedzona, dodamy ją, jeżeli nowa ścieżka jest lepsza (tańsza), niż najlepsza poprzednia ścieżka.

```
frontier = PriorityQueue()
frontier.put(start, 0)
cameFrom = {}
costSoFar = {}
cameFrom[start] = None
costSoFar[start] = 0

while not frontier.empty():
    current = frontier.get()

if current == goal:
    break

for next in graph.neighbors(current):
    newCost = costSoFar[current] + graph.cost(current, next)
    if next not in costSoFar or newCost < costSoFar[next]:
        costSoFar[next] = newCost
        priority = newCost
        frontier.pust(next, priority)
        cameFrom[next] = current
```

Użycie kolejki priorytetowej zamiast zwykłej zmienia sposób rozszerzania się granic. Kontury są jednym z sposobów, aby to zobaczyć:



Rysunek 8: Wyszukiwanie wszerz



Rysunek 9: Algorytm Dijkstry

3.3 Wyszukiwanie heurystyczne

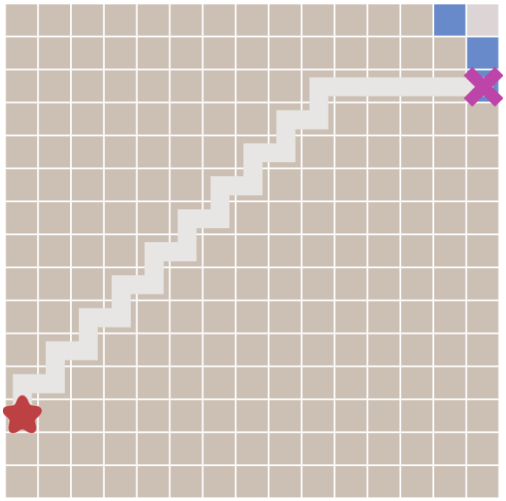
W wyszukiwaniu wszerz i algormie Dijkstry, scieżka roszerza się we wszystkich kierunkach. Jest to rozsądne rozwiązanie, jeżeli szukamy drogi do wszystkich lokalizacji. Jednakże często szukana jest ścieżka do jednej, konkretnej lokalizacji. Sprawmy, aby granica roszerzała się w strone celu bardziej, niż w innych kierunkach. Najpierw zdefiniujemy funkcję heurstyczną, która mówi nam, jak blisko celu jesteśmy:

```
def heuristic(a, b):
    return abs(a.x - b.x) + abs(a.y - b.y)
```

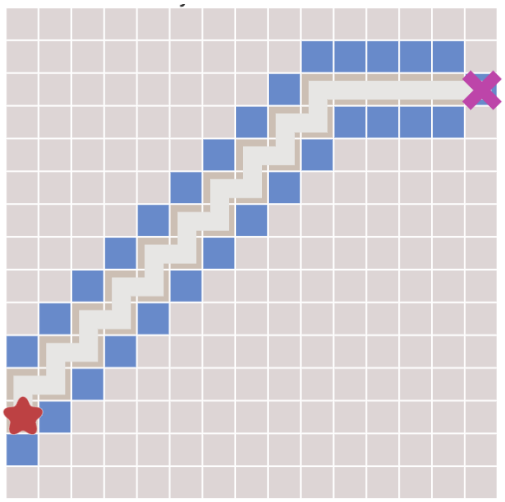
W algrytmie Dijakstry użyliśmy rzeczywistej odległości od początku do priorytetowania kolejki. Tutaj zamiast tego, w **Chciwym Wyszukiwaniu wszerz** wykorzystamy szacunkową odległość do celu dla priorytetowania kolejki. Najpierw zbadana zostanie lokalizacja najbliżej celu. Kod używa kolejki priorytetowej z alogrytmu Dijkstry, ale bez zmiennej *costSoFar*:

```
frontier = PriorytyQueue()
frontier.put(start, 0)
cameFrom = {}
cameFrom[start] = None

while not frontier.empty():
    current = frontier.get()

if current == goal:
    break

for next in graph.neighbors(current):
    if next not in cameFrom:
        priority = heuristic(goal, next)
        frontier.put(next, priority)
        cameFrom[next] = current
```

Zobaczmy jak to działa:



Rysunek 10: Wyszukiwanie wszerz



Rysunek 11: Chciwe wyszukwanie wszerz

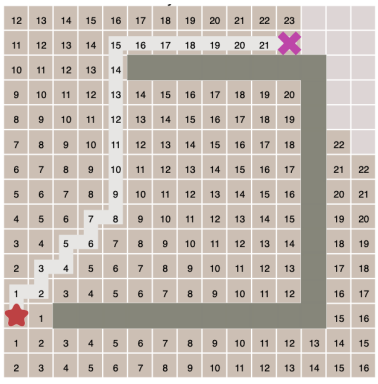
4 Algorytm A*

Algorytm Dijkstry dobrze sprawdza się w celu znalezienia najkrótszej drogi, ale marnuje czas na eksplorację w mało obiecujących kierunkach. Chciwa wersja algorytmu wyszukiwania wszerz ekslpruje droge w obiecujących kierunkach, ale nie może znaleźć najkrótszej ścieżki. Algorytm A* wykorzystuje zarówno rzeczywistą doległość od punktu, jak i szacunkową odległość do celu.

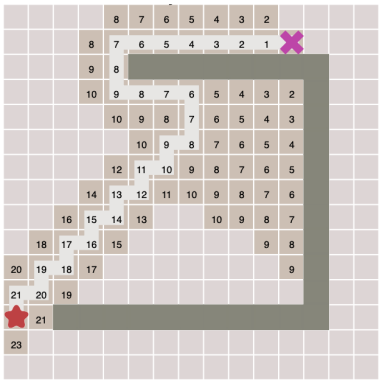
Kod jest podobny do algorytmu Dijkstry:

```
frontier = PriorytyQueue()
frontier.put(start, 0)
cameFrom = {}
costSoFar = {}
cameFrom[start] = None
costSoFar[start] = 0

while not frontier.empty():
    current = frontier.get()

    if current == goal:
        break

    for next in graph.neighbors(current):
        newCost = costSoFar[current] + graph.cost(current, next)
        if next not in costSoFar or newCost < costSoFar[next]:
            costSoFar[next] = newCost
            priority = newCost + heuristic(goal, next)
            frontier.put(next, priority)
            cameFrom[next] = current
```

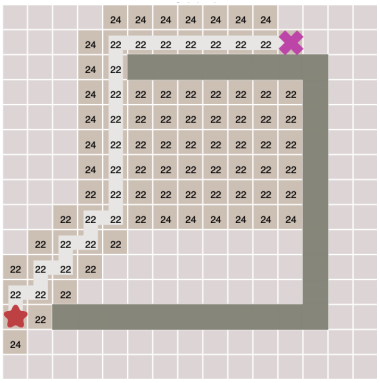
Porównując altorytmy: Algorytm Dijkstry oblicza odległość od punktu początkowego. Chciwy szacuje odległośc do punkut docelowego. A* wykorzystuje sume dwóch tych odległości.



Rysunek 12: Algorytm Dijkstry



Rysunek 13: Chciwe wyszukwanie wszerz



Rysunek 14: Algorytm A*

5 Implementacja algorytmu w języku Python

5.1 Wyszukiwanie wszerz

Pora na implementacje algorytmu wyszukiwania wszerz. Algorytm zaprezentowany w poprzednim rozdziale zawiera tylko wyszukiwanie, ale musimy również zaimplementować graf, na którym algorytm sie opiera. Oto abstrakcję, których użyje:

Graf strktura danych, która może wskazać sąsiadów każdej lokalizacji grafu. Graf wazony może również powiedzieć, jaki jest koszt przesuwania się wzdłuż krawędzi.

Lokalizacje prosta wartość (int, string, tuple, etc.), która określa lokalizacje w grafie. Nie muszą to być lokalizacje na mapie. Mogą zawierać dodatkowe informacje, takie jak kierunek, paliwo lub zapas, w zależności od rozwiązywanego problemu.

Wyszukiwanie algorytm, który na wejściu przyjmuje graf, a ściślej startową pozycję w grafie oraz opcjonalnie docelową lokalizację i oblicza użyteczne informacje (miejsca odwiedzone, dystans) dla niektórych lub wszystkich lokalizacji w grafie.

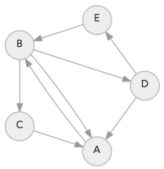
Kolejka struktura danych wykorzystywana przez algorytm wyszukiwania do decydowania o kolejności przetwarzania lokalizacji wykresów.

W poprzednim rozdziale skupiłem się na algorytmie wyszukiwania. W tym postaram się skupić na reszcie detali, aby utworzyć kompletny, działający program. Zacznę od implementacji grafu:

```
class SimpleGraph:
    def __init__(self):
        self.edges = {}

    def neighbors(self, id):
        return self.edges[id]
```

Powyższy kod to wszystko, czego potrzebujemy. Być może, ktoś zapyta gdzie jest obietk węzeł? Odpowiedź brzmi: rzadko używam obiektu węzła. Uważam, że prostsze w użyciu są inty, stringi, krotki jako lokalizacje oraz tablice lub tablice haszowane, które używają lokalizacji, jako indeksu.

Zauważ, że krawędzie są skierowane: możemy mieć krawędź od A do B, nie mając również krawędzi od B do A. W grach większośc krawędzi jest dwukierunkowa, ale czasami są drzwi jednokierunkowe lub skoki z klifów, które wyrażone są jako skierowane krawędzie. Zróbmy przykładowy graf, na którym lokalizację są literami A-E.



Rysunek 15: Przykładowy graf

Dla każdej lokalizacji potrzebuje listę miejsc, do których prowadzą:

```
example_graph = SimpleGraph()
example_graph.edges = {
    'A': ['B'],
    'B': ['A', 'C', 'D'],
    'C': ['A'],
    'D': ['E', 'A'],
    'E': ['B']
}
```

Zanim użyjemy powyższego kody razem z algorytmem wyszukiwania, musimy utowrzyć **kolejke**:

```
import collections

class Queue:
    def __init__(self):
        self.elements = collections.deque()

    def empty(self):
        return len(self.elements) == 0

    def put(self, x):
        self.elements.append(x)

    def get(self):
        return self.elements.popleft()
```

Wypróbujmy przykładowy graf z tą kolejką i kodem algorytmu wyszukiwania wszerz z głownego artykułu:

```
from implementation import *

def breathFirstSearch1(graph, start):
    frontier = Queue()
    frontier.put(start)
    visited = {}
    visited[start] = True

    while not frontier.empty():
        current = frontier.get()
        print("Visiting %r" % current)
        for next in graph.neighbors(current):
            if next not in visited:
                frontier.put(next)
                visited[next] = True

breathFirstSearch1(example_graph, 'A')
```

Wyjście programu:

```
Visiting 'A'
Visiting 'B'
Visiting 'C'
Visiting 'D'
Visiting 'E'
```

Siatki można również wyrażac jako grafy. Teraz zdefiniuje nowy **graf**, zwany SquareGrid, z **lokalizacjami** krotkami (int, int). Zamiast jawnego przechowywania krawędzi, oblicze je w funkcji sąsiadów. W wielu problemach lepiej jest je przechowywać jawnie.

```
class SquareGrid:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.walls = []

    def inBounds(self, id):
        (x, y) = id
        return 0 <= x < self.widht and 0 <= y < self.height

    def passable(self, id):
        return id not in self.walls

    def neighbors(self, id):
        (x, y) = id
        result = [(x+1, y), (x, y-1), (x-1, y), (x, y+1)]
        if (x + y) % 2 == 0: results.reverse()
        results = filter(self.inBounds, results)
        results = filter(self.passable, results)
        return results
```

Spróbujmy użyc tego, bez pierwszej siatki z poprzedniego rozdziału:

```
from implementation import *
g = SquareGrid(30, 15)
g.walls = DIAGRAM1_WALLS #long list, [(21, 0), (21, 2), ...]
draw_grid(g)
```

Wyjście:

```
. . . . . . . . . . . . . . . . . . . . . . . . . . . . ####. . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . ####. . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . ####. . . . . .
. . . ####. . . . . . . . . . . . . . . . . . . . . . . ####. . . . . .
. . . ####. . . . . . . . . ####. . . . . . . . . . . . ####. . . . . .
. . . ####. . . . . . . . . ####. . . . . . . . . . . . ##########. . . .
. . . ####. . . . . . . . . ####. . . . . . . . . . . . ##########. . . .
. . . ####. . . . . . . . . ####. . . . . . . . . . . . . . . . . . . .
. . . ####. . . . . . . . . ####. . . . . . . . . . . . . . . . . . . .
. . . ####. . . . . . . . . ####. . . . . . . . . . . . . . . . . . . .
. . . ####. . . . . . . . . ####. . . . . . . . . . . . . . . . . . . .
. . . ####. . . . . . . . . ####. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . ####. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . ####. . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . ####. . . . . . . . . . . . . . . . . . . .
```

Aby zrekonstruować scieżki, musmy zapisać lokalizacje, z której przybyliśmy, dlatego zmieniłem nazwe odwiedzone (visted) (True/False) na cameFrom (lokalizacja):

```
from implementation import *

def breathFirstSearch2(graph, start):
    # return "cameFrom"
    frontier = Queue()
    frontier.put(start)
    cameFrom = {}
    cameFrom[start] = None

    while not frontier.empty():
        current = frontier.get()
        for next in graph.neighbors(current):
            if next not in cameFrom:
                frontier.put(next)
                cameFrom[next] = current

    return cameFrom
g = SquareGrid(30, 15)
g.walls = DIAGRAM1_WALLS

parents = breathFirstSearch2(g, (8, 7))
draw_grid(g, width=2, pointTo = parents, start = (8, 7))
```

wyjście:

```
> > > > v v v v v v v v v v v v < < < < < ####v v v v v v v
> > > > > v v v v v v v v v v v < < < < < < ####v v v v v v v
> > > > > v v v v v v v v v v < < < < < < ####> v v v v v v
> > ^ ####v v v v v v v v < < < < < < < ####> > v v v v v
> ^ ^ ####> v v v v v v < ####^ < < < < < ####> > > v v v v
^ ^ ^ ####> > v v v v < < ####^ ^ < < < < ##########v v v <
^ ^ ^ ####> > > v v < < < ####^ ^ ^ < < < ##########v v < <
^ ^ ^ ####> > > A < < < < ####^ ^ ^ ^ < < < < < < < < < < <
v v v ####> > ^ ^ ^ < < < ####^ ^ ^ ^ ^ < < < < < < < < < <
v v v ####> ^ ^ ^ ^ ^ < < ####^ ^ ^ ^ ^ ^ < < < < < < < < <
v v v ####^ ^ ^ ^ ^ ^ ^ < ####^ ^ ^ ^ ^ ^ ^ < < < < < < < <
> v v ####^ ^ ^ ^ ^ ^ ^ ^ ####^ ^ ^ ^ ^ ^ ^ ^ < < < < < < <
> > > > > ^ ^ ^ ^ ^ ^ ^ ^ ####^ ^ ^ ^ ^ ^ ^ ^ ^ < < < < < <
> > > > ^ ^ ^ ^ ^ ^ ^ ^ ^ ####^ ^ ^ ^ ^ ^ ^ ^ ^ ^ < < < < <
> > > ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ####^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ < < < <
```

Niektóre implementację używają pamięci wewnetrznej, tworząc obiekt Node do przechowywania cameFrom i inne wartości dla każdego węzła grafu. Zamiast tego wybrałem użycie pamięci zewnetrznej, tworząc pojedynczą tablicę haszująca do przechowywania cameFrom dla każdego węzła grafu. Jeśli wiesz, że twoje lokalizację na mapie mają wskaźnik liczb całkowitych, inna opcją jest użycie tablicy do przechowywania cameFrom.

5.2 Wczesne wyjście

Postępując zgodnie z kodem z poprzedniego rozdziału, wystarczy dodać instrukcję if w głonwj pętli:

```
from implementation import *

def breadthFirstSearch3(graph, start, goal):
    frontier = Queue()
    frontier.put(start)
    cameFrom = {}
    cameFrom[start] = None

    while not frontier.empty():
        current = frontier.get()

        if current == goal:
            break
        for next in graph.neighbors(current):
            if next not in cameForm:
                frontier.put(next)
                cameFrom[next] = current

    return cameFrom

g = SquareGrid(30, 15)
g.walls = DIAGRAM1_WALLS

parents = breadthFirstSearch3(g, (8, 7), (17, 2))
draw_grid(g, width=2, pointTo=parents, start=(8, 7), goal=(17,2))
```

Wyjście:

```
. > > > v v v v v v v v v v v v < . . . . ####. . . . . .
> > > > > v v v v v v v v v v v < < < . . . ####. . . . . .
> > > > > v v v v v v v v v v < < < Z . . . ####. . . . . .
> > ^ ####v v v v v v v v v < < < < < < . . ####. . . . . .
. ^ ^ ####> v v v v v v < ####^ < < . . . ####. . . . . .
. . ^ ####> > v v v v < < ####^ ^ . . . . ##########. . . .
. . . ####> > > v v < < < ####^ . . . . . ##########. . . .
. . . ####> > > A < < < < ####. . . . . .
. . . ####> > ^ ^ ^ < < < ####. . . . . .
. . v ####> ^ ^ ^ ^ ^ < < ####. . . . . .
. v v ####^ ^ ^ ^ ^ ^ ^ < ####. . . . . .
> v v ####^ ^ ^ ^ ^ ^ ^ ^ ####. . . . . .
> > > > > ^ ^ ^ ^ ^ ^ ^ ^ ####. . . . . .
> > > > ^ ^ ^ ^ ^ ^ ^ ^ ^ ####. . . . . .
. > > ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ####. . . . . .
```

Widać, że algorytm zatrzymuje się, gdy znajdzie cel Z.

5.3 Algorytm Dijkstry

Agotytm ten dodaje złożoności do wyszukiwania grafów, ponieważ zaczniemy przetwarzać lokalizację w lepszej kolejnośći, niż "pierwsze wchodzi, pierwsze wychodzi"(FIFO). Co musimy zmienić?

- Graf musi znać koszt ruchu.
- Kolejka musi zwrócić węzły w innej kolejności.
- Wyszukiwanie musi śledzić te koszty na grafie i przekazywać je do kolejki.

5.3.1 Graf z wagami

Zwykły graf pokazuje sąsiadów każdego węzła. Graf ważony mówi również o kosztach przemieszczania się wzdłuż każdej krawędzi. Dodam funkcje *cost(fromNode, toNode)*, która mówi nam o kosztach przemieszczania się z lokalizacji *fromNode* do jego sąsiada *toNode.*W tej leśnej mapie postanowiłem, że ruch zależy tylko od *toNode*:

```
class GridWithWeights(SquareGrid):
    def __init__(self, width, height):
        super().__init__(width, height)
        self.weights = {}

    def cost(self, fromNode, toNode):
        return self.weights.get(toNode, 1)
```

5.3.2 Kolejka priorytetowa

Kolejka priorytetowa kojaży każdy element z numerem zwanym "priorytetem". Zwracając przedmiot, wybiera ten o najniższym numerze.

Oto dość szybka kolejka priorytetowa, która wykorzystuje kopiec binarny, ale nie obsługuje zmiany priorytetu. Aby uzyskać prawidłową obsługę, wykorzystamy krotki (priorytet, element).

```
import heapq

class PriorityQueue:
    def __init__(self):
        self.elements = []

    def empty(self):
        return len(self.elements) == 0

    def put(self, item, priority):
        heapq.heappush(self.elements, (prioirty, item))

    def get(self):
        return heapq.heappop(self.elements)[1]
```

5.3.3 Wyszukiwanie

Oto trudna kwestia dotycząca implementacji: po dodaniu kosztów ruchu można ponownie odwiedzić lokalizację z lepsza *costSoFar*. To znaczy, że linijka *if next not in cameFrom* nie będzie działać. Zamiast tego muszę srpawdzić, czy koszt spadł od ostatniej wizyty.

```
def dijkstraSearch(graph, start, goal):
    frontier = PriorityQueue()
    frontier.put(start, 0)
    cameFrom = {}
    costSoFar = {}
    cameFrom[start] = None
    costSoFar[start] = 0

    while not frontier.empty():
        current = frontier.get()

        if current == goal:
            break

        for next in graph.neighbors(current):
            newCost = costSoFar[current] + graph.cost(current, next)
            if next not in costSoFar or newCost < costSoFar[next]:
                costSoFar[next] = newCost
                priority = newCost
                frontier.put(next, priority)
                cameFrom[next] = current

    return cameFrom, costSoFar
```

Po implementacji wyszukiwania muszę zbudować scieżkę:

```
def reconstructPath(cameFrom, start, goal):
    current = goal
    path = []
    while current != start:
        path.append(current)
        current = cameFrom[current]
    path.append(start)
    path.reverse()
    return path
```

Chociaż scieżki najlepiej traktować jako sekwencje krawędzi, wygodnie jest przechowywać je jako sekwencję węzłów. Aby zbudować ścieżkę, zacznij od końca i postępuj zgodnie z *cameFrom* map, który wskazuje na poprzedni węzeł. Kiedy dojdziemy do początku, jesteśmy skończeni. Jest to ścieżka wstecz, więc trzeba wywołać funkcje *reverse()* na końcu *reconstructPath* jeśl potrzebujesz przechowywać go dalej. Czasami wygodniej jest przechowywać wstecz. Czasami warto również zapisać węzeł początkowy na liście. Pora wypróbować kod:

```
from implementation import *
cameFrom, costSoFar = dijkstraSearch(diagram4, (1, 4), (7, 8))
drawGrid(diagram4, width=3, pointTo=cameFrom, start=(1, 4), goal=(7, 8))
print()
drawGrid(diagram4, width=3, number=costSoFar, start=(1, 4), goal=(7, 8))
print()
drawGrid(diagram4, width=3, path=reconstructPath(cameFrom, start=(1, 4), goal=(7,
    8))
```

Wyjście programu:

```
v   v   <   <   <   <   <   <   <   <
v   v   <   <   <   ^   ^   <   <   <
v   v   <   <   <   <   ^   ^   <   <
v   v   <   <   <   <   <   ^   ^   .
>   A   <   <   <   <   .   .   .   .
^   ^   <   <   <   <   .   .   .   .
^   ^   <   <   <   <   <   .   .   .
^   #########^   <   v   .   .   .
^   #########v   v   v   Z   .   .
^   <   <   <   <   <   <   <   <   .

5   4   5   6   7   8   9   10  11  12
4   3   4   5   10  13  10  11  12  13
3   2   3   4   9   14  15  12  13  14
2   1   2   3   8   13  18  17  14  .
1   A   1   6   11  16  .   .   .   .
2   1   2   7   12  17  .   .   .   .
3   2   3   4   9   14  19  .   .   .
4   #########14  19  18  .   .   .
5   #########15  16  13  Z   .   .
6   7   8   9   10  11  12  13  14  .

.   .   .   .   .   .   .   .   .   .
.   .   .   .   .   .   .   .   .   .
.   .   .   .   .   .   .   .   .   .
.   .   .   .   .   .   .   .   .   .
@   @   .   .   .   .   .   .   .   .
@   .   .   .   .   .   .   .   .   .
@   .   .   .   .   .   .   .   .   .
@   #########.   .   .   .   .   .
@   #########.   .   @   @   .   .
@   @   @   @   @   @   @   .   .   .
```

5.4 Algorytm A*

Zarówno chciwe wyszukiwanie wszerz, jak i A* używaja funkcji heurystycznej. Jedyna róznica polega na tym, że A* wykorzystuje zarówno heurystykę, jak i kolejność z algorytmu Dijkstry. Oto algorytm A*:

```
def heuristic(a, b):
    (x1, y1) = a
    (x2, y2) = b
    return abs(x1 - x2) + abs(y1 - y2)

def aStarSearch(graph, start, goal):
    frontier = PriorityQueue()
    frontier.put(start, 0)
    cameFrom = {}
    costSoFar = {}
    cameFrom[start] = None
    costSoFar[start] = 0

    while not frontier.empty():
        current = frontier.get()

    if current == goal:
        break

    for next in graph.neighbors(current):
        newCost = costSoFar[current] + graph.cost(current, next)
        if next not in costSoFar or newCost < costSoFar[next]:
            costSoFar[next] = newCost
            priority = newCost + heuristic(goal, next)
            frontier.put(next, priority)
            cameFrom[next] = current

    return cameFrom, costSoFar
```

Wypróbujmy algorytm:

```
from implementation import *
sratr, goal = (1, 4), (7, 8)
cameFrom, costSoFar = aStarSearch(diagram4, start, goal)
drawGrid(diagram4, width = 3, pointTo=cameFrom, start=start, goal=goal)
print()
drwaGrid(diagram4, width=3, number=costSoFar, start=start, goal=goal)
print()
```

wyjscie:

```
.   .   .   .   .   .   .   .   .   .
.   v   v   v   .   .   .   .   .   .
v   v   v   v   <   .   .   .   .   .
v   v   v   <   <   .   .   .   .   .
>   A   <   <   <   .   .   .   .   .
>   ^   <   <   <   .   .   .   .   .
>   ^   <   <   <   <   .   .   .   .
^   #########^   .   v   .   .   .
^   #########v   v   v   Z   .   .
^   <   <   <   <   <   <   <   .   .

.   .   .   .   .   .   .   .   .   .
.   3   4   5   .   .   .   .   .   .
3   2   3   4   9   .   .   .   .   .
2   1   2   3   8   .   .   .   .   .
1   A   1   6   11  .   .   .   .   .
2   1   2   7   12  .   .   .   .   .
3   2   3   4   9   14  .   .   .   .
4   #########14  .   18  .   .   .
5   #########15  16  13  Z   .   .
6   7   8   9   10  11  12  13  .   .
```