Złożoność obliczeniowa

Złożoność obliczeniowa BFS.
$G = (V, E)$.

1. Stała ilość operacji do zwiedzenia wierzchołka, bez odkrywania jego sąsiadów.
2. Odwiedzając wierzchołek badamy wszystkie jego sąsiady (i być może odkładamy je do kolejki): stała ilość operacji na sąsiada.
3. Odwiedzin wierzchołków jest tyle, co wierzchołków, czyli $|V|$.
4. Badań sąsiadów jest tyle, co krawędzi w grafie, czyli $|E|$.

Stąd: złożoność czasowa: $O(|V| + |E|)$.
Złożoność pamięciowa: $\Theta(|V|)$.

Złożoność obliczeniowa algorytmu Dijsktry.
Zależy od implementacji kolejki priorytetowej. U nas:
Analiza podobna jak dla BFS, ale zamiast wykonywać stałą ilość operacji przy odwiedzaniu wierzchołka, korzystamy z kopca. Kopiec przechowuje wszystkie wierzchołki i operacje na nim są w $O(\log |V|)$.
Zatem złożoność czasowa: $O\big((|E| + |V|) \log |V|\big)$.
Złożoność pamięciowa: $\Theta(|V|)$

Dla kopców Fibonacciego: złożoność czasowa $\mathrm{decrease\_min}$ jest stała, co poprawia złożoność czasową algorytmu do $O\big(|E| + |V| \log |V|\big)$.

Przeszukiwanie wszerz i algorytm Dijkstry mają podobny schemat:

- W kolejnych krokach odwiedzamy wierzchołki - zawsze odwiedzany jest sąsiad jednego z wierzchołków odwiedzonych wcześniej.
- Odwiedzając wierzchołek, robimy coś z nim i z jego sąsiadami.
- Mamy sposób, który decyduje o kolejności odwiedzania.

BFS - odwiedzając wierzchołek, „odkrywamy" jego nieodkryte sąsiady, kolejność odwiedzania to kolejność odkrywania. Używamy kolejki do śledzenia kolejności.

Algorytm Dijkstry - kolejność odwiedzana jest decydowana przez odległości tymczasowe, uaktualnianie w sąsiadach podczas odwiedzin wierzchołka. Do śledzenia tych odległości używamy kolejki priorytetowej.

A **inne algorytmy** oparte o ten schemat?

Znajdowanie najkrótszej ścieżki

W BFS i algorytmie Dijkstry, rozpoczynamy trawersowanie od wybranego wierzchołka-źródła, rekonstruując najkrótsze ścieżki do wszystkich wierzchołków z niego osiągalnych.

Bardzo często jednak interesuje nas jedynie najkrótsza ścieżka od źródła do konkretnie wybranego wierzchołka. W obu algorytmach, ścieżka taka jest skonstruowana dokładnie w momencie odwiedzania tego wierzchołka.

W konsekwencji, zamiast prowadzić algorytm do końca, można przerwać jego działanie w momencie osiągnięcia docelowego wierzchołka i zwrócić zrekonstruowaną ścieżkę, nie odwiedzając niepotrzebnie potencjalnie wielkiej liczby wierzchołków.

Mimo to, oba algorytmy muszą „niepotrzebnie" odwiedzić pewną ilość wierzchołków, które nie leżą na znalezionej ścieżce. Jak zminimalizować tę ilość?

Algorytm A*

Czasami potrafimy powiedzieć coś o odległości między dwoma wierzchołkami w grafie - na przykład łatwo znaleźć jej oszacowanie **z dołu**.

Niech $G = (V, E, f)$ będzie grafem skierowanym dodatnio ważonym. Dla $x, y \in V$ oznaczmy przez $d(x, y)$ odległość z x do y (długość najkrótszej ścieżki z x do y, czyli sumę wag jej krawędzi).

Funkcję $h : V \times V \to \mathbb{R}_{\geqslant 0} \cup \{\infty\}$ nazwiemy **dopuszczalną, spójną funkcją heurystyczną** (w skrócie: **heurystyką**), gdy spełnia następujące warunki:

1. (dopuszczalność) $h(x, y) \leqslant d(x, y)$ dla wszystkich $x, y \in V$,
2. (spójność) $h(x, z) \leqslant d(x, y) + h(y, z)$ dla wszystkich $x, y, z \in V$.

Algorytm A*

Algorytm A* - algorytm (właściwie: klasa algorytmów) znajdowania najkrótszej ścieżki między parą wierzchołków w grafie skierowanym dodatnio ważonym, „ulepszający" algorytm Dijkstry.

Wejściem dla algorytmu jest graf skierowany dodatnio ważony wraz z parą wierzchołków (*źródła* u i *celu* v) **oraz heurystyką** h pozwalającą na szacowanie odległości między wierzchołkami.

Jego kroki są identyczne z algorytmem Dijkstry (w wersji, która przerywa działanie po osiągnięciu celu) z następującą różnicą: zamiast wybierać do odwiedzin wierzchołek x o najmniejszej możliwej odległości tymczasowej $g(x)$, wybieramy ten, dla którego minimalną wartość ma suma $g(x) + h(x, v)$.

Inaczej mówiąc: preferujemy odwiedzanie tych wierzchołków, których suma *przeszacowanej* odległości od źródła i *niedoszacowanej* odległości do celu jest najmniejsza możliwa.

Fakt. Gdy h jest dopuszczalna i spójna, to algorytm A* jest **dopuszczalny**, czyli poprawnie wyznacza (pewną) najkrótszą ścieżkę od źródła do celu.

Uwaga. Dla każdego grafu, dopuszczalnymi i spójnymi heurystykami zawsze są:

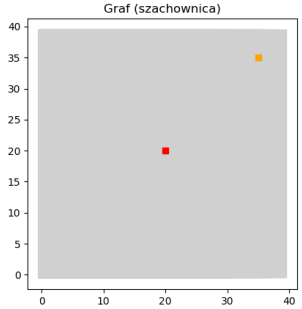
- Funkcja stała równa 0. Wtedy algorytm jest identyczny z algorytmem Dijkstry.
- Funkcja $h(x, y) = d(x, y)$, czyli prawdziwa odległość (rzadko kiedy mamy do dyspozycji taką „wyrocznię").

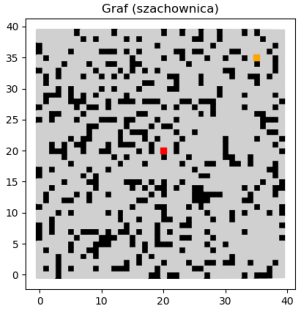
Dobranie dobrej heurystyki pozwala na minimalizację niepotrzebnie odwiedzanych wierzchołków. Zobaczmy to na przykładzie grafów-szachownic, które często pojawiają się w symulacjach i grach (planszowych i komputerowych).

Algorytm A*

Rozważmy planszę $n \times m$ składającą się z pól jasnych i pól czarnych. Możemy poruszać się po planszy, ale tylko po polach jasnych, i tylko pomiędzy polami sąsiadującymi ze sobą bokami.

Chcemy znaleźć najkrótszą ścieżkę ze źródła (czerwone pole) do celu (pomarańczowe pole). Zakładamy, że ścieżka istnieje.





Algorytm A*

Planszę możemy traktować jako graf. Jego wierzchołkami są pola jasne, ich kluczami są „współrzędne“ pola. Krawędzie kładziemy między wierzchołkami reprezentującymi sąsiednie pola. Wagi wszystkich krawędzi to 1.

Jest to wtedy graf nieskierowany dodatnio ważony, więc można na nim wykonać algorytmy BFS, Dijsktry lub A* - ten ostatni po ustaleniu heurystyki.

Szczęśliwie, mamy tu wiele heurystyk do wyboru.

Uwaga. Interesują nas jedynie heurystyki, które da się wyliczyć szybko (najlepiej w czasie stałym).

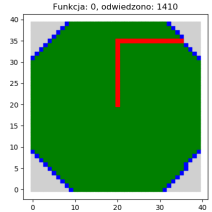
Algorytm A*

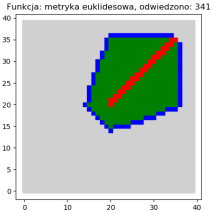
Przykłady heurystyk dla grafów-szachownic - każdy wierzchołek utożsamiamy z jego współrzędnymi.

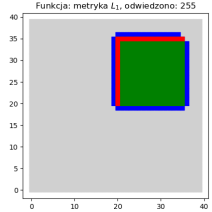
- Funkcja stała równa 0 (zawsze dobra). Wtedy algorytm „degeneruje się" do algorytmu Dijkstry, a ponieważ wszystkie wagi krawędzi są równe 1, algorytm Dijkstry degeneruje się do BFS.
- Metryka euklidesowa.
- Metryka L_∞ (Czebyszewa).
- Metryka L_1 (taxicab).
- Odległość rzutów punktów na oś OX (lub OY).
- ...

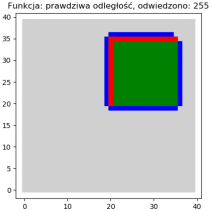
Algorytm A*

A* dla różnych heurystyk na planszy bez czarnych pól (pola zielone: odwiedzone, pola niebieskie: choć raz relaksowane):



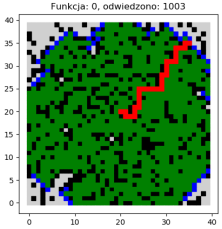


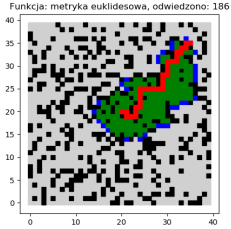


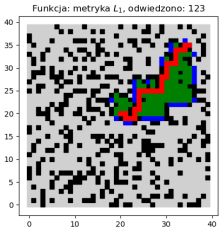


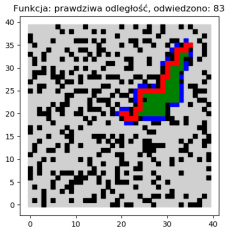
Algorytm A*

To samo, gdy pewne zostały zaczernione (w pewien losowy sposób):









Obserwujemy, że im „lepsza" heurystyka (lepiej szacująca prawdziwą odległość), tym mniej wierzchołków zostaje odwiedzanych przy konstrukcji ścieżki. Odzwierciedla to następujące:

Stwierdzenie: Jeśli $h_1(x, y)$ i $h_2(x, y)$ są dopuszczalnymi i spójnymi funkcjami heurystycznymi, i dla wszystkich $x, y \in V$ zachodzi $h_1(x, y) \leqslant h_2(x, y)$, to A* używający h_1 odwiedzi co najmniej te same wierzchołki, co A* używający h_2.

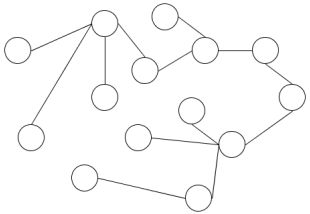
Minimalne drzewo rozpinające

Pewna terminologia:

- Graf nieskierowany w którym pomiędzy każdą parą wierzchołków znajdziemy ścieżkę nazwiemy **spójnym**.
- **Drzewem nieukorzenionym** nazwiemy spójny graf nieskierowany bez cykli (może być ważony lub nie).

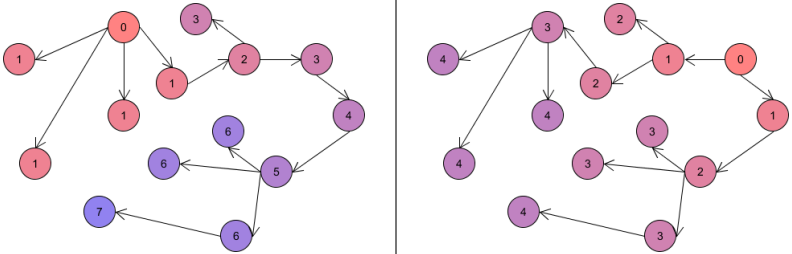
Drzewo nieukorzenione może być puste. Każda para wierzchołków w drzewie nieukorzenionym jest połączona dokładnie jedną ścieżką, w której krawędzie się nie powtarzają.

Przykład:



Minimalne drzewo rozpinające

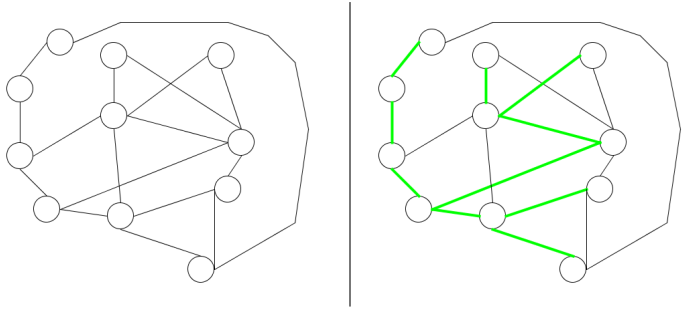
W nieukorzenionym drzewie nie wyróżniamy korzenia. Możemy jednak wybrać dowolny wierzchołek i przemienić drzewo nieukorzenione w drzewo („zwykłe", czyli **ukorzenione**) o korzeniu w tym wierzchołku:



Na obrazkach zaznaczone są odległości od wybranego wierzchołka-korzenia (w szczególności korzeń to jedyny wierzchołek z liczbą 0). Odległości determinują kierunek relacji rodzic-dziecko.

Minimalne drzewo rozpinające

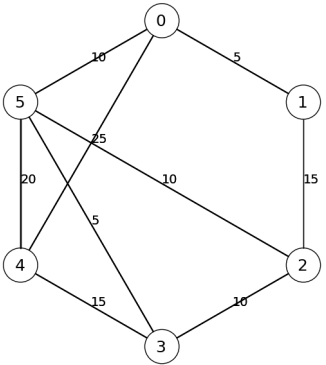
Niech $G = (V, E)$ będzie nieskierowanym grafem spójnym. **Drzewem rozpinającym** w G nazwiemy dowolny graf $G' = (V, E')$, gdzie $E' \subseteq E$, który jest drzewem nieukorzenionym.
(Inaczej: G' to podgraf G o tych samych wierzchołkach, który jest drzewem ukorzenionym). Przykładowo:

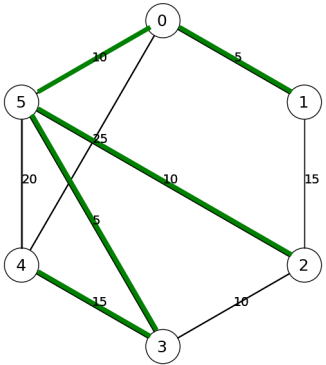


Analogicznie dla grafów ważonych (spójnych i nieskierowanych).

Minimalne drzewo rozpinające

Niech $G = (V, E, f)$ będzie teraz nieskierowanym grafem spójnym nieskierowanym ważonym. **Minimalnym drzewem rozpinającym** (*minimum spanning tree*, *MST*) w G nazwiemy takie drzewo rozpinające, w którym suma wag krawędzi jest najmniejsza możliwa. Przykładowo:





Minimalne drzewo rozpinające

MST mają wiele zastosowań (np. planowanie optymalnych połączeń sieci komputerowej, machine learning - analiza skupień, genetyka, wizualizacja danych).

Dwa klasyczne algorytmy znajdowania minimalnego drzewa rozpinającego: Prima i Kruskala, z których omówimy ten pierwszy.

Algorytm Prima

W algorytmie Prima, drzewo konstruujemy w krokach, dodając do niego nowe wierzchołki wraz z krawędziami, dołączając je do rozrastającego się drzewa. Robimy to w sposób *zachłanny*, wybierając krawędzie o jak najmniejszej wadze.

Kroki algorytmu Prima dla spójnego grafu ważonego (niekoniecznie dodatnio) $G = (V, E, f)$.

1. Wybierz dowolny wierzchołek $v \in V$. Stwórz drzewo $T = (\{v\}, \varnothing)$.
2. Dopóki są w grafie wierzchołki, które nie są w T:
   1. Wybierz krawędź e łączącą wierzchołek x z drzewa T z wierzchołkiem y spoza drzewa T o najmniejszej możliwej wadze.
   2. Dodaj do drzewa y i krawędź e.

Algorytm Prima

Fakt. Algorytm Prima jest poprawny.

Implementacja algorytmu Prima: znów modyfikujemy algorytm Dijkstry, w następujący sposób:

1. Odwiedzając wierzchołek t, do relaksowania odległości tymczasowej jego sąsiada s nie używamy sumy:

   $$(\text{odległość tymczasowa } t) + (\text{waga krawędzi z } t \text{ do } s),$$

   a jedynie wagi krawędzi z t to s.
2. Po wykonaniu algorytmu, krawędzie poprzedników węzłów to wynik: są krawędziami minimalnego drzewa rozpinającego.

Algorytm Prima

W czasie wykonania takiej modyfikacji algorytmu Dijkstry:

1. Wierzchołki odwiedzone to wierzchołki z konstruowanego drzewa T.
2. Modyfikacja z punktu 1 gwarantuje, że odległość tymczasowa każdego sąsiada co najmniej jednego wierzchołka z tych już odwiedzonych jest równa odległości od najbliższego z nich; poprzednik pamięta, która krawędź zaświadcza o tej odległości.
3. Wybór nieodwiedzonego wierzchołka o najmniejszej odległości tymczasowej jest tożsamy z wyborem wierzchołka spoza drzewa połączonego z drzewem najkrótszą możliwą krawędzią.

Złożoność obliczeniowa

Złożoności (czasowa i pamięciowa) algorytmu A* oraz algorytmu Prima: takie, jak dla algorytmu Dijkstry.

(Dla A* zakładamy tutaj, że funkcja heurystyczna działa w czasie $O(\log |V|)$ w wersji dla kopców binarnych, lub $O(1)$ w wersji dla kopców Fibonacciego).

Przeszukiwanie w głąb

Algorytm DFS (*depth-first search*, *przeszukiwanie w głąb*) - inny sposób przeszukiwania grafu, w którym wierzchołki odkryte odwiedzamy w kolejności **odwrotnej** do kolejności ich odkrywania. Dopuszczamy wielokrotne odkrywanie tego samego wierzchołka, ale każdy może zostać odwiedzony co najwyżej raz.

Kroki visit (v - wierzchołek w grafie) zapisane rekurencyjnie:

1. Oznacz v jako odwiedzony.
2. Dla każdego nieodwiedzonego w, sąsiada v: wykonaj rekurencyjnie visit na w.

W odróżnieniu od BFS, w którym wierzchołki były odwiedzane w kolejności odległości od źródła, DFS odwiedza wierzchołki leżące na pewnej ścieżce dopóki nowoodwiedzony wierzchołek ma nieodwiedzonego sąsiada, i cofa się (wychodząc z rekurencji) dopiero, gdy takiego sąsiada nie ma.

Przeszukiwanie w głąb

Dwie implementacje DFS:

```
def dfs(v): # Rekurencyjna: jak w opisie
    visited = set()
    def visit(w):
        visited.add(w)
        for u in w.get_connections():
            if u not in visited: visit(u)
    visit(v)

def dfs(v): # Iteracyjna: stos zamiast stosu wywolan
    visited = set()
    stack = Stack()
    stack.push(v)

    while not stack.is_empty():
        w = stack.pop()
        if w in visited: continue
        visited.add(w)
        for u in w.get_connections():
            if u not in visited: stack.push(u)
```

Przeszukiwanie w głąb

Z przyczyn praktycznych preferujemy implementację iteracyjną. Wersja rekurencyjna może powodować problemy - przekroczenie dopuszczalnej głębokości rekurencji.

Obie implementacje możemy modyfikować, wykonując pewne działania, gdy odwiedzamy wierzchołek. Przykładowo: możemy oznaczać krawędzie prowadzące do sąsiada, którego odwiedzamy w kroku następnym - podobnie jak oznaczanie „poprzednika" przy odkrywaniu wierzchołka w BFS.

Wersję rekurencyjną łatwiej zmodyfikować, niż iteracyjną.

Przeszukiwanie w głąb

Nieodwiedzone sąsiady odwiedzanego wierzchołka możemy odwiedzać w dowolnej kolejności.

Rozważmy graf-szachownicę rozmiaru $n \times m$, na którym wszystkie pola są „wolne". W takim grafie, skoro wierzchołek utożsamiamy ze współrzędnymi pola, można mówić o sąsiedzie „lewym", „górnym", etc.

Przeszukiwanie w głąb

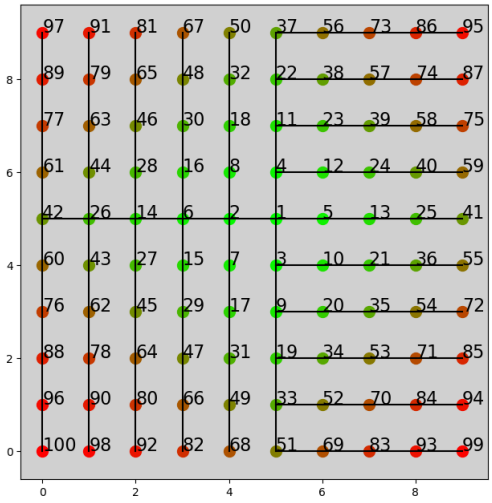
Na grafie-szachownicy 10×10 przeprowadźmy przeszukiwanie grafu na trzy sposoby, rozpoczynając z wierzchołka $(5, 5)$:

1. BFS, w którym mamy preferowaną kolejność odkrywania wierzchołków: lewy, dolny, górny, prawy
2. DFS, gdzie wybierając sąsiada do dalszych odwiedzin, preferujemy kolejno: prawy, górny, dolny, lewy.
3. DFS, gdzie sąsiad do dalszych odwiedzin jest wybierany losowo (spośród nieodwiedzonych, jeśli istnieją).

Na dalszych obrazkach, każde pole-wierzchołek otrzyma numer (i kolor) zgodnie z kolejnością odwiedzania. Dla BFS, połączenia są zgodne z zapamiętaną relacją poprzednika. Dla DFS, kładziemy krawędź między wierzchołkiem a odwiedzonym z niego sąsiadem.

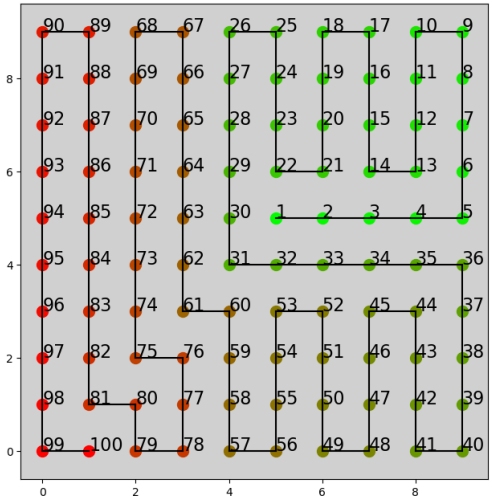
Przeszukiwanie w głąb

BFS:



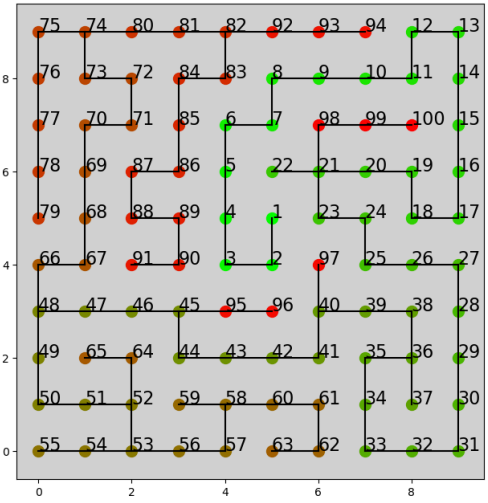
Przeszukiwanie w głąb

DFS z ustaloną preferencją:



Przeszukiwanie w głąb

DFS z preferencją losową:



Przeszukiwanie w głąb

Obserwujemy:

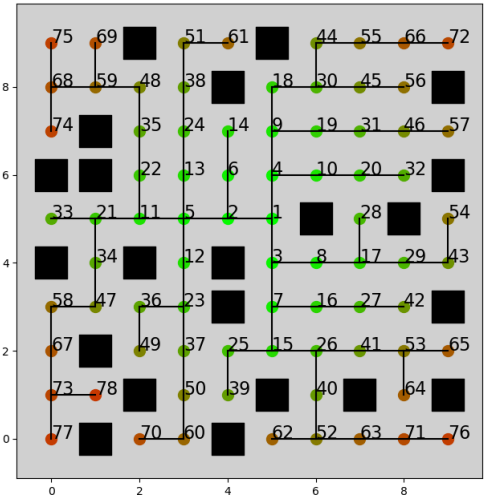
1. Krawędzi dołączyliśmy tyle, ile było odwiedzonych wierzchołków minus 1; wszystkie wierzchołki są połączone krawędzią. Zawsze otrzymujemy zatem drzewo rozpinające.
2. W DFS z preferencją, wyszła pojedyncza ścieżka (maksymalne głębokość rekurencji).
3. W DFS z losowaniem pojawia się ciekawe drzewo: interpretując wierzchołki jako „komnaty" otoczone ścianami, i połączenia uzyskane przez DFS jako „drzwi", otrzymujemy labirynt. Z każdej komnaty do innej prowadzi dokładnie jedna ścieżka (bez cofania się).

Odnośnie 3: podobnie będzie dla trzech (lub więcej) wymiarów.

Spójrzmy też na efekty algorytmów na planszach, w których część pól jest czarna (wykreślona).

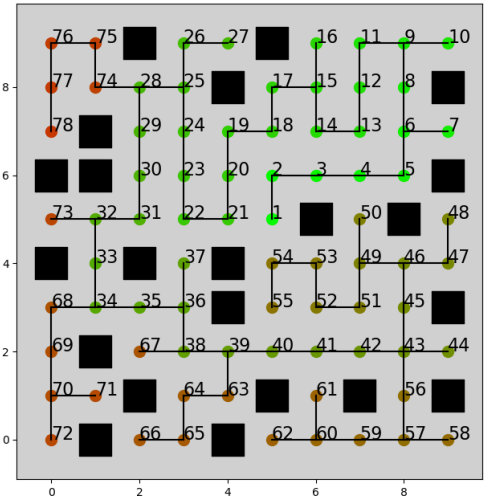
Przeszukiwanie w głąb

BFS:



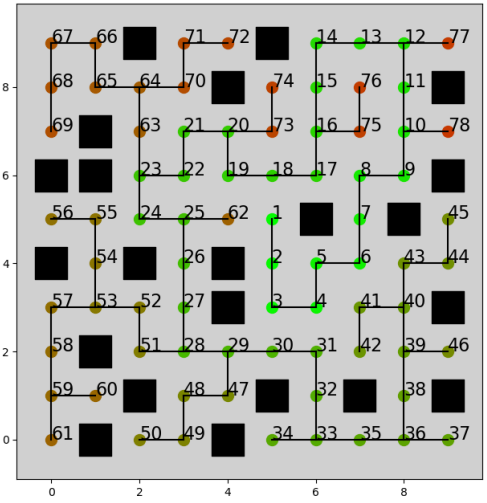
Przeszukiwanie w głąb

DFS z ustaloną preferencją:



Przeszukiwanie w głąb

DFS z preferencją losową:



Sortowanie topologiczne

Zastosowanie DFS: sortowanie topologiczne.

Niech $G = (V, E)$ będzie skierowany i bez cykli (tak zwany **acykliczny graf skierowany**, *directed acycylic graph*, *DAG*). Graf taki można utożsamić z diagramem Hassego relacji częściowego porządku.

Fakt. Każdy porządek częściowy można rozszerzyć do porządku liniowego.

Problem: uporządkować liniowo wierzchołki w G w sposób kompatybilny z krawędziami grafu (czyli rozszerzyć relację E do porządku liniowego).

Sortowanie topologiczne

W sortowaniu topologicznym, wykonujemy DFS począwszy od różnych węzłów w grafie, wykonując pewne dodatkowe operacje po odwiedzeniu wierzchołka. Wierzchołki mają trzy „stany" - nieodwiedzone (białe), tymczasowo odwiedzone (szare), oraz trwale odwiedzone (czarne).

Kroki $\mathrm{topological\_sort}$ ($G = (E, V)$ - graf acykliczny skierowany).

1. Stwórz pustą listę L.
2. Dopóki w G znajduje się nieodwiedzony wierzchołek:
   1. Wybierz dowolny nieodwiedzony wierzchołek $v \in V$.
   2. Rozpocznij DFS z wierzchołka v, oznaczając odwiedzane wierzchołki jako tymczasowo odwiedzone, ale nie odwiedzaj wierzchołków trwale odwiedzonych (czarnych). Po zakończeniu zwiedzania wszystkich sąsiadów $w \in V$, oznacz v jako trwale odwiedzony i odłóż go na listę L.
3. Zwróć elementy z L w odwrotnej kolejności.