Algorytmy grafowe

Mateusz Bednarski (11794), Nikodem Hynek (117209)

1 Wstęp

Celem pracy jest implementacja dwóch podstawowych metod reprezentacji grafu. Przez listę następników oraz macierz sąsiedztwa. Dla każdej z tych reprezentacji zaimplementowano dwa algorytmy: przechodzenie w głąb i wszerz. Daje to w sumie cztery algorytmy:

- BFS list
- DFS list
- BFS matrix
- DFS matrix

2 Metodyka testów

Algorytmy zostały przygotowane w języku Python (2.7.6), wersja 64-bit. Komputer na którym odbyły się pomiary: 8 GB RAM, Intel i5 3,1GHz pracujący pod kontrolą systemu Windows 8.1 (x64). Wykresy zostały wygenerowane przy użyciu bibliotek *matplotlib* oraz *prettyplotlib* Sprawozdanie powstało przy użyciu systemu składu tekstu LaTeX. Wielkości próbek (n) dla każdego algorytmu to 11, 72, 143, 215, 286, 357, 429, 500, 571, 643, 714, 785, 857, 928 oraz 1000 elementów (liczb całkowitych nieujemnych). Każdy punkt wykresu pokazuje średnią ze 1000 pomiarów oraz odchylenie standardowe.

3 Lista następników

Reprezentacja grafu w postaci listy następników polega na utworzeniu słownika (tablicy asocjacyjnej/mapy) której kluczami są węzły a wartościami listy następników tychże węzłów.

3.1 BFS

Algorithm 1: BFS_list

Data: graph — tablica asocjacyjna grafu
start — wierzchołek startowy
path[] — lista kolejno odwiedzonych wierzchołków

```
begin
    stack ← utwórz listę stack.append(start)
    while start jest niepusty do
        current ← stack.pop()
        if ¬ current ∈ path then
            path.append(current) stack ← graph[current] + stack
        end
    end
    return path
end
```

Złożoność algorytmu wynosi $\mathcal{O}(|V| + |E|)$.

3.2 DFS

Algorithm 2: DFS_list

```
Data:  graph — tablica asocjacyjna grafu
start — wierzchołek startowy
path[] — lista kolejno odwiedzonych wierzchołków
begin
    queue ← utwórz kolejkę queue.enqueue(start)
    while queue jest niepusta do
        current ← queue.dequeue()
        if ¬ current ∈ path then
            path.append(current)
            zakolejkuj wszystkich nieodwiedzonych następników.
        end
    end
    return path
end
```

Złożoność algorytmu wynosi $\mathcal{O}(|V| + |E|)$.

4 Macierz sąsiedztwa

Implementacja w oparciu o macierz sąsiedztwa jest bardzo prosta. Tworzymy macierz o rozmiarze $n \times n$ i wypełniamy ją zerami. Teraz jeżeli istnieje łuk między węzłami a i b to ustawiamy matrix[a][b] = 1.

4.1 DFS

Algorithm 3: DFS_matrix

```
Data:  n — ilość węzłów
discovered[n] — tablica typu bool zainicjowana wartością false
v — wierzchołek od którego zaczynamy przejście.
begin
    print(v)
    discovered[v] ← true
    foreach w będącego sąsiadem v do
        if ¬ discovered[w] then
            DFS_matrix(w)
        end
    end
end
```

Złożoność algorytmu wynosi $\mathcal{O}(|V| + |E|)$.

4.2 BFS

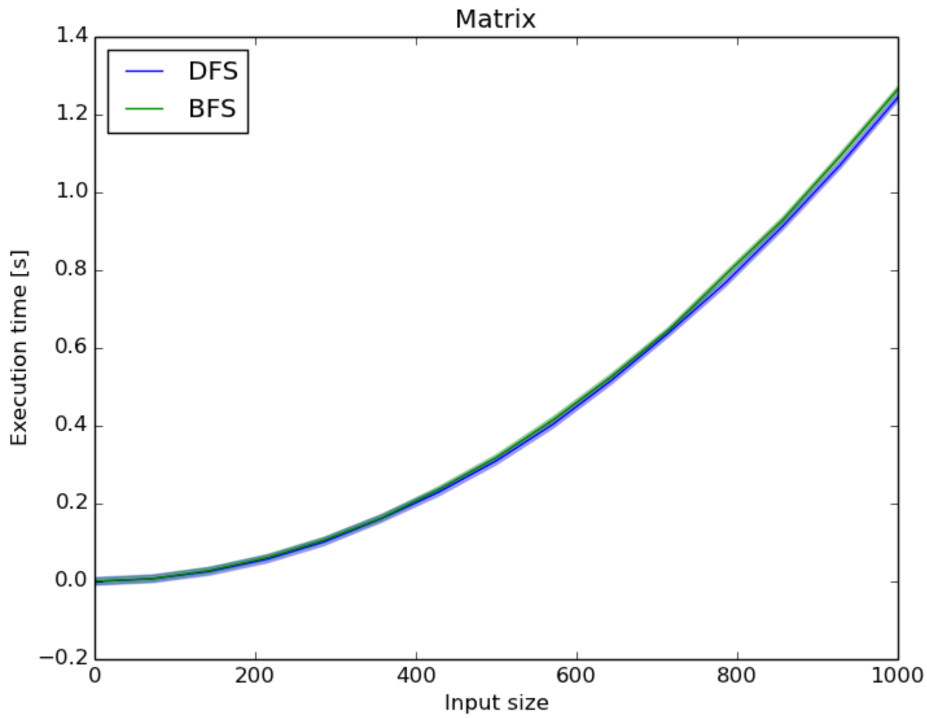
Algorithm 4: BFS_matrix

Data: n — ilość węzłów
BLACK, GRAY, WHITE — stałe
s — wierzchołek od którego rozpoczyna się przechodzenie
color, dist, parent — tablice n-wymiarowe

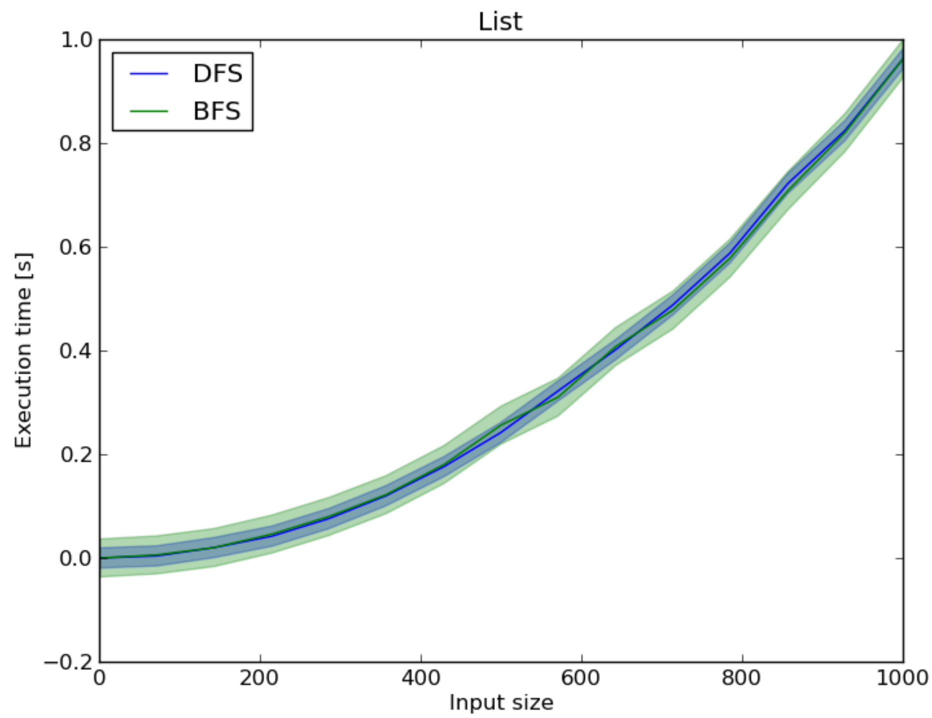
```
begin
    zainicjuj tablicę color wartością WHITE
    zainicjuj tablicę dist wartością +∞
    zainicjuj tablicę color wartością null
    color[s] ← GRAY
    dist[s] ← 0
    parent[s] ← null
    q ← zainicjuj kolejkę
    q.enqueue(s)
    while q jest niepusta do
        u ← q.dequeue()
        foreach v będącego sąsiadem u do
            if color[v] = WHITE then
                color[v] = GRAY
                dist[v] = dist[u] + 1
                parent[v] = u
            end
        end
        color[u] ← BLACK
        print(u)
    end
end
```

Złożoność algorytmu wynosi $\mathcal{O}(|V| + |E|)$.

5 Testy



Rysunek 1: Wykres $t(n)$ dla algorytmów *BFS_matrix* i *BFS _matrix*. Widać że zachowują się praktycznie tak samo.



Rysunek 2: Wykres $t(n)$ dla algorytmów *BFS_list* i *BFS _list*. Złożoność jest taka sama.

6 Bonus

Zadanie bonusowe polegało na zaproponowaniu algorytmu znajdowania ścieżek. Postawiony problem brzmi następująco: Posiadasz plan pewnego muzeum – jest ono w formie kwadratu podzielonego na n * n równych pomieszczeń (n – dowolna liczba całkowita). W każdym pomieszczeniu możliwa jest tylko jedna z trzech następujących sytuacji: zwiedzający oglądają znajdujące się tam dzieła sztuki, w pomieszczeniu znajduje się strażnik lub pomieszczenie jest puste. Twoim zadaniem jest znaleźć ścieżkę z pomieszczenia A do pomieszczenia B (o ile to możliwe) taką, że składa się ona z pustych pomieszczeń możliwe najbardziej odległych od strażników. Przyjmij, że liczba strażników wynosi m, natomiast przechodzenie z pomieszczenia do pomieszczenia możliwe jest tylko horyzontalnie i wertykalnie. Dodatkowo możesz przyjść że punkt A jest jednym z pomieszczeń przy brzegu budynku (wyjście z muzeum), wówczas punkt B może oznaczać pomieszczenie z najbardziej wartościowym dziełem sztuki. Przyjmujemy, że pomieszczenia A i B są puste.

Kolejne sekcje szczegółowo omawiają sposób rozwiązania opracowany przez autorów.

6.1 Dane wejściowe

Algorytm przyjmuje następujące parametry wejściowe:

n Wielkość mapy (bok kwadratu)

m Procent strażników

vp Procent pokoi zajętych przez zwiedzających

startPos Miejsce startowe

treasurePos Lokacja najbardziej wartościowego łupu

guard_range Zasięg strażnika

6.1.1 Zasięg strażnika

Ostatni parametr wejściowy wymaga szerszego komentarza. W przyjętym rozwiązaniu każdy strażnik roztacza wokół siebie pole wektorowe. Jego wyższa wartość oznacza większe prawdopodobieństwo zauważenia intruza. Zasięg oznacza maksymalną odległość (w metryce taksówkowej) do której sięga generowane przez niego pole. Sposób generowania pola zostanie omówiony przy algorytmach pomocniczych.

6.2 Sposób reprezentacji danych

Problem jest reprezentowany przez macierz o rozmiarze $n \times n$. Każda komórka zawiera liczbę całkowitą kodującą jej stan. Możliwe wartości to:

-5 Przez pole przebiega znaleziona ścieżka

-4 Pole zawierające łup

-3 Pole startowe

-2 Pole zawierające strażnika

-1 Pole zawierające zwiedzającego

0+ Pole puste, wartość oznacza „koszt” przejścia (tym większy im bliżej są strażnicy)

Wraz z algorytmem powstał program rysujący mapę.

6.3 Pole generowane przez strażnika

Każdy strażnik roztacza wokół siebie pole wektorowe. Do generowania zastosowano dwie funkcje, z czego jedna z nich została ostatecznie użyta[1].

6.4 Algorytmy pomocnicze

Na potrzeby głównego algorytmu powstało kilka mniejszych (pomocniczych).

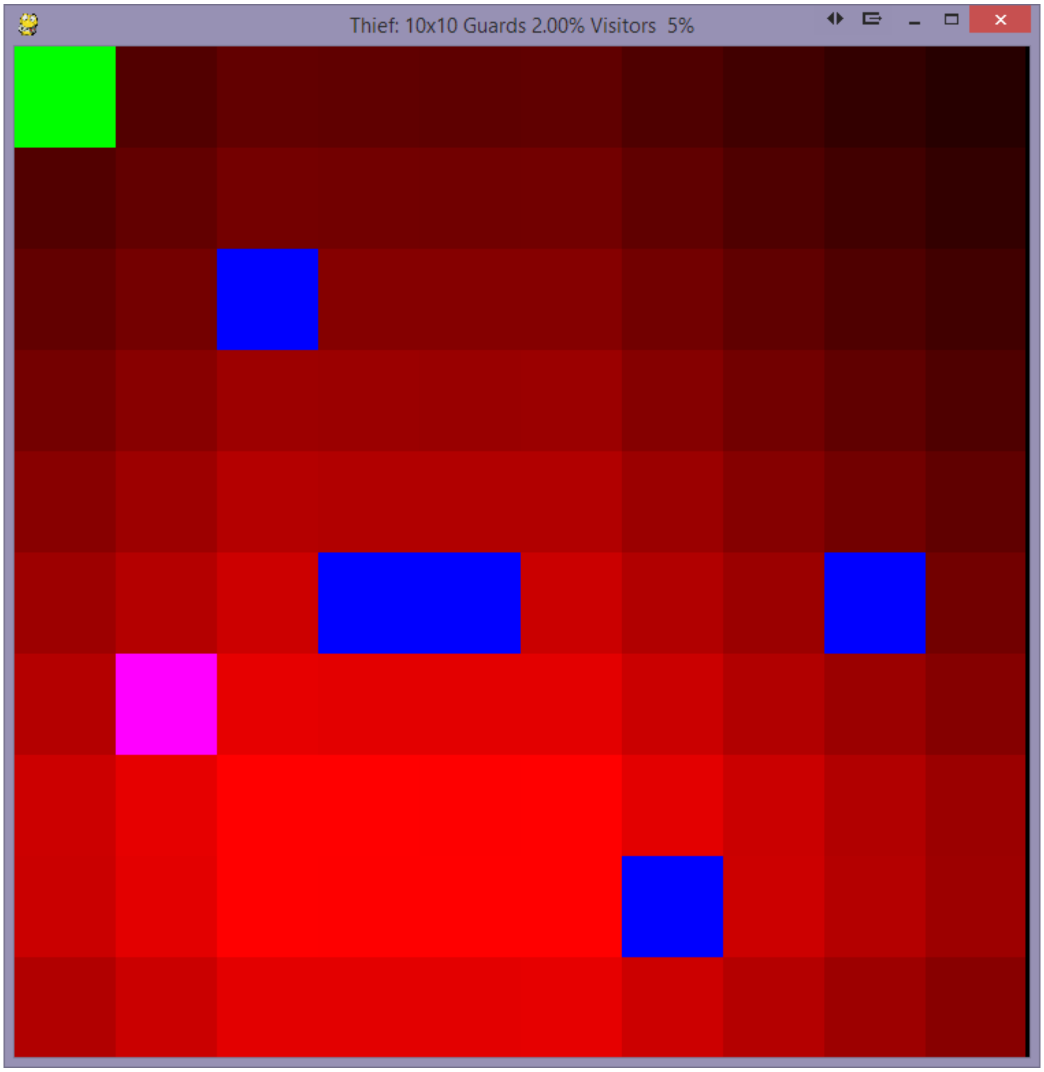
6.4.1 Odległość w sensie metryki taksówkowej

Algorithm 5: distance

Data: $\vec{a}$, $\vec{b}$ — współrzędne dwóch punktów
Result: Odległość
begin
  return $|x_a - x_b| + |y_a - y_b|$
end

Złożoność alorytmu wynosi $\mathcal{O}(1)$.

[1]Drugą funkcją jest *linear_cost* w pliku *museum.py*



Rysunek 3: Mapa wygenerowana dla parametrów $n = 10$, $m = 2\%$, vp = 10%, startPos = $[0, 0]$, treasurePos = $[6, 1]$, guard range = 20
Kolor zielony oznacza miejsce startowe. Różowy — łup, niebieski — zwiedzającego. Nasycenie czerwonego — wartość pola skalarnego. Kolory są skalowane liniowo tak aby #FF0000 oznaczał maksymalną wartość pola na całej mapie a #000000 wartość zerową. Wartość #FF0000 oznacza również strażnika.

$$J(x) = \begin{cases} 0 & \text{dla } x = 0 \lor x > C_1 \\ \frac{C_1^2}{C_2}(x - C_1)^2 + 1 & \text{dla } x \in (0, C_1) \end{cases}$$

Kwadratowa funkcja kosztu. Argumentem jest odległość pola od strażnika. C_1 oraz C_2 oznaczają odpowiednio zasięg strażnika oraz bazowy koszt przejścia (na polu ze strażnikiem). W praktyce wielkość C_2 współczynnika jest nieistotna, byłoby inaczej gdyby istniały klasy strażników o różnej wartości np. strażnik z bronią miał by tą wartość większą, niż ochroniarz z grupą inwalidzką)

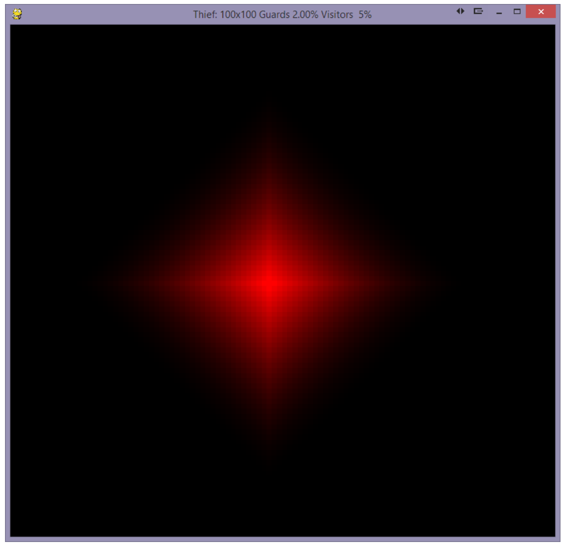
6.4.2 Dodanie pola pochodzącego od nowego strażnika

Algorithm 6: append_field

Data: pos — współrzędne strażnika, n — rozmiar mapy
Result: Odległość
`/* gr oznacza zasięg strażnika */`

begin
  foreach *pos.y - gr* $< y <$ *pos.y + gr* **do**
    foreach *pos.x - gr* $< x <$ *pos.x + gr* **do**
      if $0 \leq x \leq n \land 0 \leq y \leq n \;\; x$ **then**
        if $[x, y]$ *nie jest zajęty* **then**
          cost $\leftarrow J(\text{dist}(pos, [y, x])$
          matrix[y][x] += cost
        end
      end
    end
  end
end

Złożoność wynosi $\mathcal{O}(n^2)$ gdzie n jest zasięgiem strażnika.



Rysunek 4: Pole generowane przez pojedynczego strażnika.

6.4.3 Rozmieść strażników

Algorithm 7: put_guards

Data: n — ilość strażników

`/* is_prohibited sprawdza czy na danym polu można umieścić strażnika tj. czy nie zawiera ono już innego obiektu */`

begin
    put ← 0
    while $put < n$ **do**
        pos ← wylosuj pozycję na planszy
        if $\neg is\_prohibited(pos)$ **then**
            ustaw pole *pos* w macierzy na -2 (strażnik)
            append_field(pos)
            put += 1
        end
    end
end

Algorytm dba o to aby zostało rozmieszczone *dokładnie* n strażników. Jeśli wylosowane pole jest już zajęte, próbuje dalej. W szczególnym przypadku algorytm nigdy się nie kończy (strażników do rozmieszenia jest więcej niż wolnych pól). Złożoność określamy jako $\mathcal{O}(n^3)$ (ze względu na 7 linię) chociaż dla małych plansz z dużą ilością strażników będzie ona rosła ze względu na ponawianie losowań. Ustawianie zwiedzających jest analogicznie, jedyną różnicą jest wstawianie -1 w miejsce -2 więc nie jest ono tutaj prezentowane.

6.5 Generowanie mapy

Algorytm generowania mapy należy zasadniczo do algorytmów pomocniczych, jednak ze względu na swoją wagę zostanie omówiony osobno.

Algorithm 8: Konstruktor klasy *Museum*

Data: n — Wielkość mapy (bok kwadratu)
m — Procent strażników
vp — Procent pokoi zajętych przez zwiedzających
sp — Miejsce startowe
tp — Lokacja najbardziej wartościowego łupu
gr — Zasięg strażnika

```
1 begin
2     zainicjuj macierz matrix o rozmiarze n × n
3     matrix[s_p] ← -3
4     matrix[t_p] ← -4
5     gc ← n^2 · m ÷ 100
6     vc ← n^2 · vp ÷ 100
7     put_guards(g_c)
8     put_visitors(v_c)
9 end
```

Algorytm jest bardzo prosty i nie wymaga specjalnego komentarza. Przyjrzyjmy się jednak jego złożoności. W tabeli przedstawiono złożoność instrukcji w kolejnych liniach:

2	3	4	5	6	7	8
$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(g_c \cdot g_r^2)$	$\mathcal{O}(\text{vc})$

Daje to $\mathcal{O}(5 + g_c \cdot g_r^2 + v_c)$. Następnie

$$\mathcal{O}\left(5 + n^2 \cdot m \cdot g_r^2 + n^2 \cdot v_p\right)$$

Przy racjonalnym założeniu, że $m \approx \text{vp}$

$$\mathcal{O}\left(5 + n^2 \cdot m \cdot g_r^2 + n^2 \cdot v_p\right)$$

Ostatecznie

$$\mathcal{O}\left(n^2(mg_r^2 + v_p)\right)$$

Wyliczona złożoność jest dość wysoka. Jest to spowodowane doliczaniem wartości generowanych przez pole dookoła strażnika. Jak się jednak później okaże się że taka reprezentacja danych jest efektywna przy wyszukiwaniu drogi.

6.6 Znajdowanie drogi

Zaproponowany algorytm znajdowania drogi został w całości wymyślony przez autorów. Został zaimplementowany w dwóch wersjach: normalnej i „nie niszczącej”. Druga wersja nie zaznacza znalezionej drogi i została przygotowana na potrzeby pomiarów.

Algorytm pracuje w dwóch fazach, które zostaną omówione osobno.

6.6.1 Budowanie macierzy kosztu

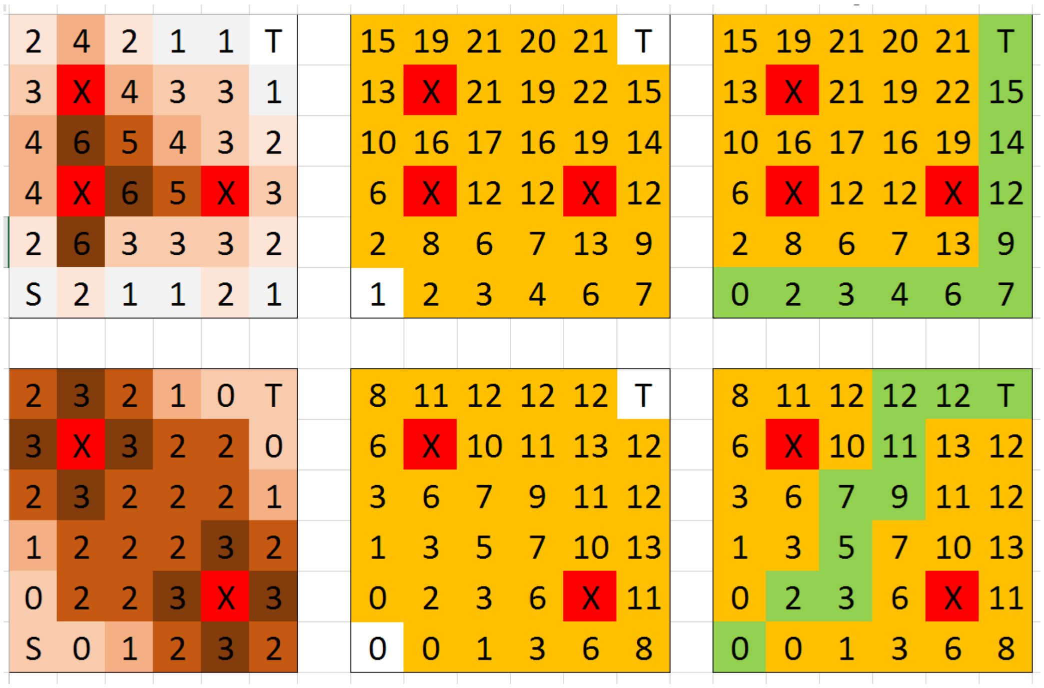
Faza pierwsza polega na przejściu całej mapy i wpisaniu w każde pole minimalnego kosztu przejścia do niego. Zaczynamy od pola startowego i wszędzie dookoła wrzucamy koszt obecnego pola + 1 + koszt pola sąsiedniego. Dodawanie jedynki sprawia że spośród kilku ścieżek o tym samym sumarycznym koszcie algorytm preferuje krótszą. Jako że do jednego pola można dojść z kilku stron, algorytm wybiera tą o najniższym koszcie.

Faza kończy się po dotarciu do łupu. W takim przypadku część tablicy *hard* może pozostać nieuzupełniona. Nie jest to problemem ponieważ wszystkie koszty są dodatnie, więc wiadomo już że nie da się znaleźć „tańszej” ścieżki.

Złożoność algorytmu szacujemy na $\mathcal{O}(n^2)$

Algorithm 9: steal (I)

```
Data: n — Wielkość mapy (bok kwadratu)
sp — Miejsce startowe
matrix — wygenerowana plansza
Result: hard — macierz kosztów osiągnięcia każdego pola
/* is_way - czy przez pole można przejść                          */
begin
    hard ← zainicjuj macierz n × n wartością +∞
    visited ← zainicjuj macierz n × n wartością false
    q ← utwórz kolejkę
    q.enqueue(sp)
    while q jest niepusta do
        curr ← q.dequeue()
        visited[curr] ← true
        if matrix[curr] = -4 then
            /* Znaleziono łup - przeszliśmy wystarczająco
               dużo                                               */
            return hard
        end
        foreach nb ∈ pola_sąsiednie_do _curr do
            if ¬ is_way(nb) ∨ visited[nb] then
                continue
            end
            newCost ← 1 + hard[curr] + matrix[nb]
            oldCost ← hard[nb]
            hard[nb] = min(newCost, oldCost)
            if nb ∉ q then
                q.enqueue(nb)
            end
        end
        visited[curr] ← true
    end
end
```



Rysunek 5: Macierz z danymi i macierze kosztu dla dwóch przykładowych danych. Z lewej widać zawartość macierzy *matrix* z zawartymi wartościami pola skalarnego. Macierz w środku prezentuje *hard* (bez uwzględnienia dodatkowych jedynek). Widać, że im dalej od punktu startowego koszt dotarcia na pole rośnie. Macierze z prawej są również reprezentacją *hard* ale z zaznaczonymi znalezionymi ścieżkami.

6.6.2 Właściwe znajdowanie drogi

Mając w ten sposób przygotowaną macierz kosztu, znalezienie optymalnej drogi jest bardzo proste.

Zaczynamy w miejscu gdzie znajduje się łup. Następnie przechodzimy na sąsiednie pole o najmniejszym koszcie i oznaczamy je jako drogę [2]. Algorytm postępuje aż natrafi na jeden z dwóch przypadków:

1. Nie ma pola na które można przejść - oznacza to brak drogi między punktem startowym a eksponatem

[2] Istnieje jeszcze druga wersja algorytmu *no_destroy* która pomija ten krok - pozwala to na wielokrotne użycie tej samej macierzy w testach, bez konieczności generowania jej od nowa.

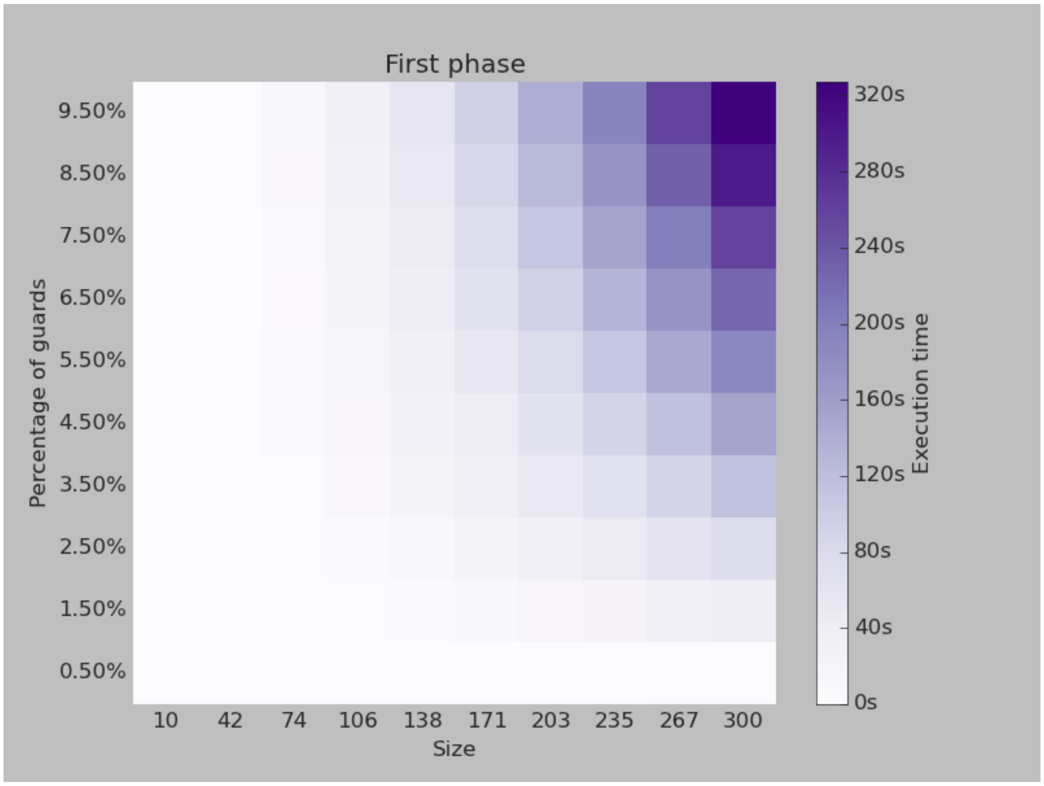
2. dotarcie do punktu startowego - oznacza to osiągnięcie celu

Złożoność szacujemy na $\mathcal{O}(n)$.

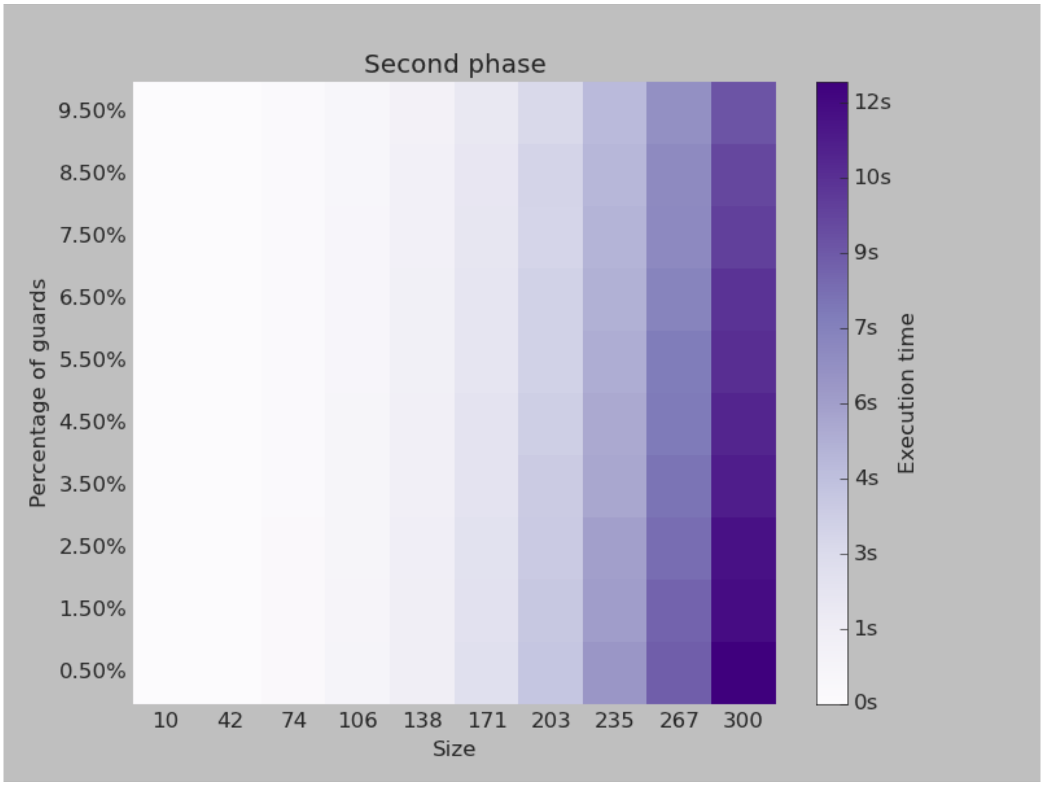
Algorithm 10: steal (II)

```
Data:  n — Wielkość mapy (bok kwadratu)
sp — Miejsce startowe
matrix — wygenerowana plansza
target — lokacja łupu
hard — macierz kosztu przejścia
Result: hard — macierz kosztów osiągnięcia każdego pola
/* is_way - czy przez pole można przejść                    */
begin
    visited ← zainicjuj macierz n × n wartością 0
    iter ← target
    while true do
        if iterator = null then
            print „No Way!”
            break
        end
        if iterator = sp then
            break
        end
        visited[iter] ← true
        matrix[iter] ← -5 ;                 /* Oznacz jako drogę */
        local_minima ← +∞
        next_way ← null
        foreach nb ∈ pola_sąsiednie_do ∧ is_way(nb) do
            val ← hard[nb]
            if val < local_minima then
                local_minima ← val
                next_way ← nb
            end
        end
        iterator ← next_way
    end
end
```

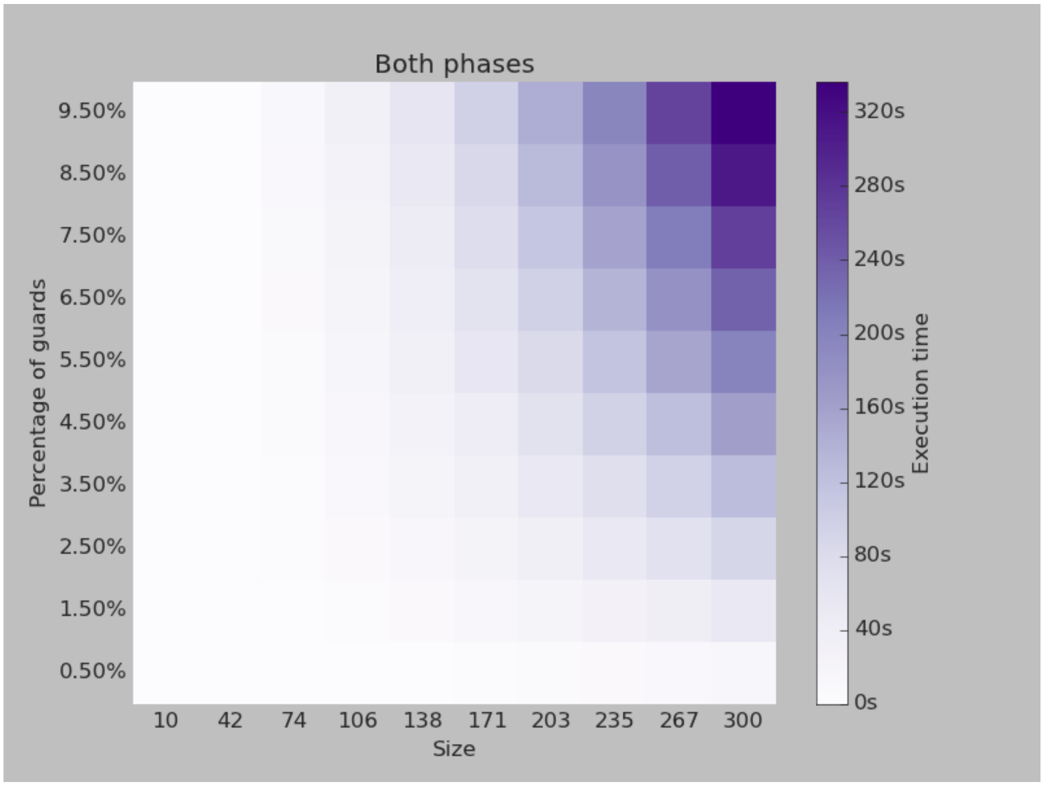
6.7 Analiza



Rysunek 6: Wykres utworzony dla pierwszej części algorytmu. Widać, że im większa gęstość strażników oraz wielkość mapy tym dłuższy czas generowania. Wpływ czynnika g_r i n są bardzo podobne, zgodnie z teoretycznymi przewidywaniami.



Rysunek 7: Wykres utworzony dla drugiej części algorytmu. Decydująca jest tutaj wielkość mapy (co również zgadza się z teorią). Wpływ ilości strażników jest niewielki i co ciekawe im jest ich więcej tym szybciej działa algorytm (przeciwnie niż w pierwszej fazie).



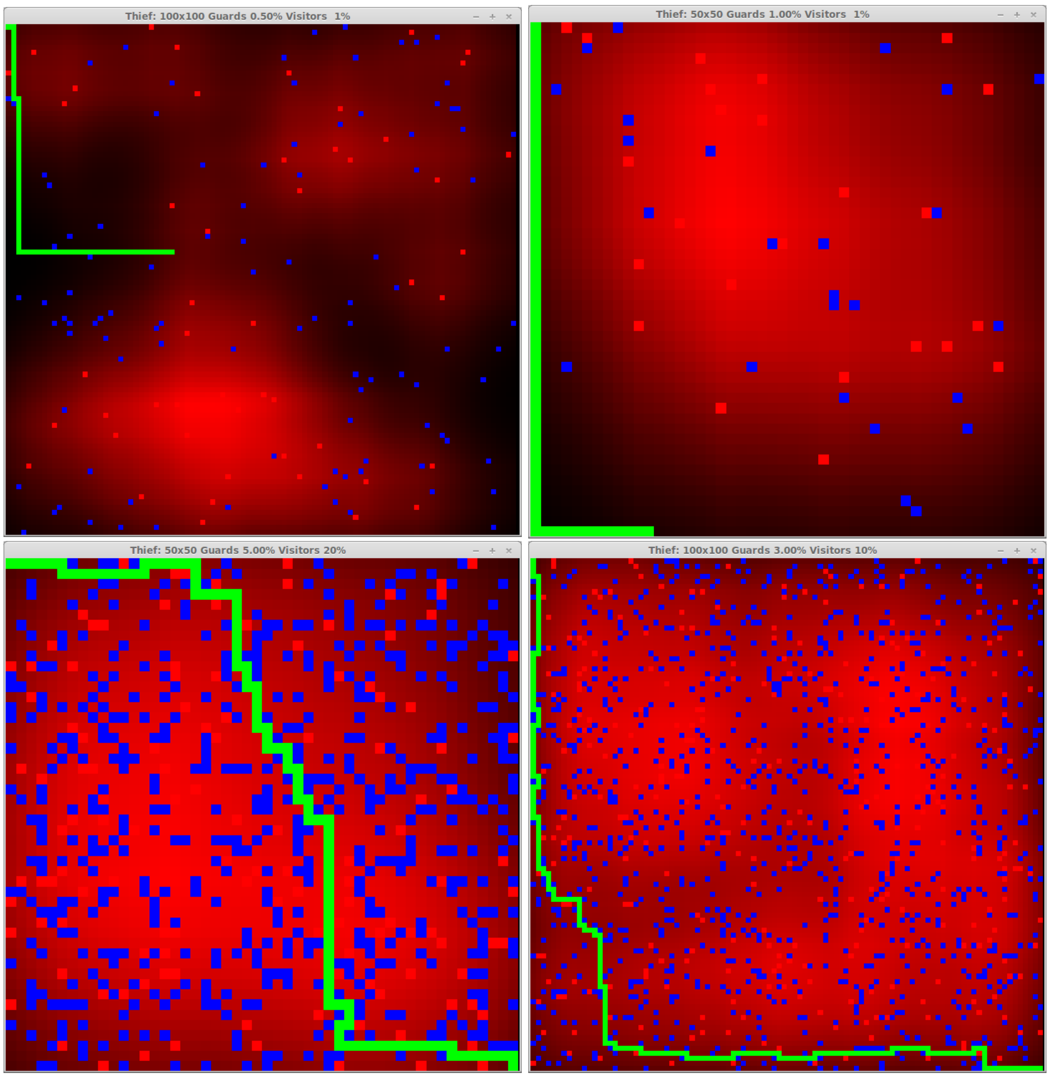
Rysunek 8: Nałożone czasy wykonania obu faz. Jest on całkowicie zdominowany przez pierwszą fazę.

6.8 Podsumowanie

Autorzy początkowo rozważali użycie jednego ze znanych algorytmów m. in. *Floyda—Warshalla*, *Dijkstry* oraz *A**. Ostatecznie uznali że realizacja własnego pomysłu będzie bardziej wartościowa niż rozwiązanie tego problemu tak samo po raz kolejny. Ponadto przejście przez cały proces od wymyślenia do zaimplementowania i przetestowania ma większą wartość edukacyjną.

Cały kod jest dostępny w serwisie Github - `https://github.com/Xevaquor/AiSD-Graph`

Algorytm łatwo da się rozszerzyć o dynamiczne przesuwanie strażników, jednakże nie wystarczyło na to czasu.



Rysunek 9: Przykładowe wyniki działania algorytmu.