SZKOŁA GŁÓWNA HANDLOWA W WARSZAWIE
WARSAW SCHOOL OF ECONOMICS

Porównanie efektywności algorytmów przeszukiwania przestrzeni z wykorzystaniem środowiska gry Pacman.

Projekt zaliczeniowy z przedmiotu
Modelowanie wieloagentowe [234900-0286]

Mikołaj Jaworski, 63548

Warszawa, 2019

Spis treści

Wstęp

Celem niniejszej pracy było porównanie efektywności działania algorytmów przeszukiwania przestrzeni w środowisku gry Pacman. Implementacji poddano cztery rodzaje rozwiązań: *depth-first search*, *breadth-first search*, *uniform-cost search*, *A* search* wraz z heurystyką. Funkcje były wykorzystywane do znalezienia ścieżki do kropek (pożywienia Pacmana) w różnych konfiguracjach: ich ilości oraz układu i rozmiaru środowiska-labiryntu.

Praca została podzielona na dwie części. W pierwszej części pracy zawarto krótki opis działania wykorzystanych algorytmów. W drugiej przedstawiono proces przeprowadzania testów oraz ich rezultaty, gdzie głównym miernikiem była szybkość znajdywania rozwiązania (mierzona ilością sprawdzonych węzłów w grafie możliwych ścieżek). Zawarto również konsolowe komendy do wywoływania programu, aby Czytelnik mógł sprawdzić samodzielnie funkcjonalności.

Kod źródłowy gry Pacman wraz z funkcjami pomocniczymi pochodzi z materiałów udostępnionych w ramach kursu *CS188: Introduction to AI* na Uniwersytecie Kalifornijskim. Cała aplikacja, a co za tym idzie również implementowane funkcje zostały napisane w języku programowania Python 2.7. Zaimplementowane funkcje znajdują się w plikach *search.py* oraz *searchAgents.py.* Aby wywołać program, należy użyć komend podanych w komentarzach w odpowiednich sekcjach.

Własności algorytmów

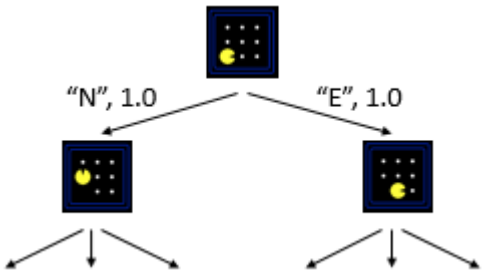
Algorytmy z kategorii *tree search* (lub *tree traversal*) stosujemy do konkretnych problemów, w których przeszukujemy graf różnych ścieżek w poszukiwaniu rozwiązania, posiadających ważne elementy:

1. *State space*, czyli zbiór wszystkich możliwych konfiguracji środowiska (pozycje Pacmana oraz pożywienia). Każdy stan w nim występuje tylko raz. Zwykle nie jesteśmy w wstanie go zbudować i zachować w pamięci komputera ze względu na jego rozmiary.
2. *Successor function*, za pomocą której definiowane są możliwych przejść między stanami (aktualizowanie pozycji oraz macierzy pożywienia).
3. Stany początkowe oraz celu (brak pożywienia na całej przestrzeni).

Zwracanym rezultatem jest sekwencja akcji prowadząca do rozwiązania.

W drzewach przeszukujących przestrzeń stany mogą występować wielokrotnie. W tym przypadku również zwykle nie tworzy się pełnego drzewa ze względu na ograniczenia obliczeniowe, jak również dlatego, że w niektórych przypadkach drzewo może mieć nieskończoną głębokość. Ważnym pojęciem związanym z drzewem jest *fringe,* czyli zestaw węzłów, które bierzemy pod uwagę do eksploracji na podstawie zbudowanych do tej pory ścieżek. W zależności od stosowanego algorytmu, jej zawartość może się znaczaco różnić.

Grafika 1. Początek drzewa przeszukującego.



Źródło: materiały do wykładów CS188, UC Berkeley.

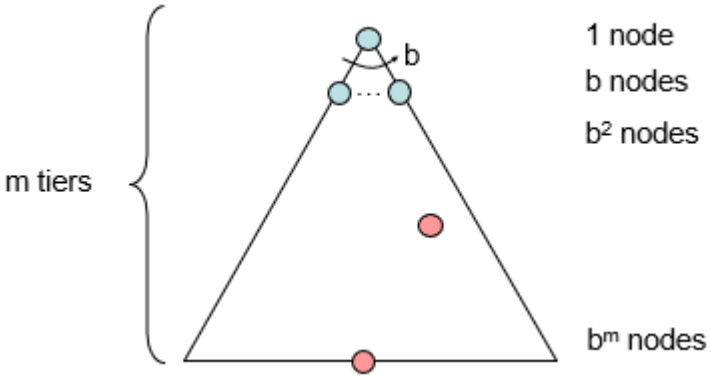
Algorytmy możemy podzielić na dwie kategorie: *uninformed* oraz *informed.* Do pierwszej kategorii możemy zaliczyć pierwsze trzy z opisanych w kolejnych podrozdziałach rozwiązania. Z tej listy tylko A* należy do drugiej kategorii ze względu na użycie heurestyk. Tak czy inaczej oparte są na takim samym schemacie:

```
funkcja przeszukiwania(problem, strategia):
        zdefiniuj stan początkowy z określonego problemu
        powtarzaj:
                jeżeli nie ma możliwości dlaszego badania przestrzeni, zwróć []
                wybierz węzeł ze zbioru możliwych do sprawdzenia na podstawie strategii
                jeżeli sprawdzany węzeł jest taki sam jak stan docelowy, zwróć rozwiąznie
                w innym przypadku rozwiń węzeł dalej i dodaj kolejne węzły do zbioru do sprawdzenia
        zwróć rozwiązanie
```

Do opisu algorytmów stosowane są pojęcia takie jak:

1. Kompletność – czy zagwarantowane jest znalezienie rozwiązania?
2. Optymalność – czy jest pewne, że algorytm znajdzie najkrótsze możliwe rozwiązanie?
3. Złożoność czasowa – jak długo zajmie znalezienie rozwiązania.
4. Złożoność przestrzenna – jak dużo węzłów należy przetrzymywać we *fringe*.

Grafika 2. Złożoność drzew przeszukujących, gdzie m – głębokość drzewa, b – ilość węzłów

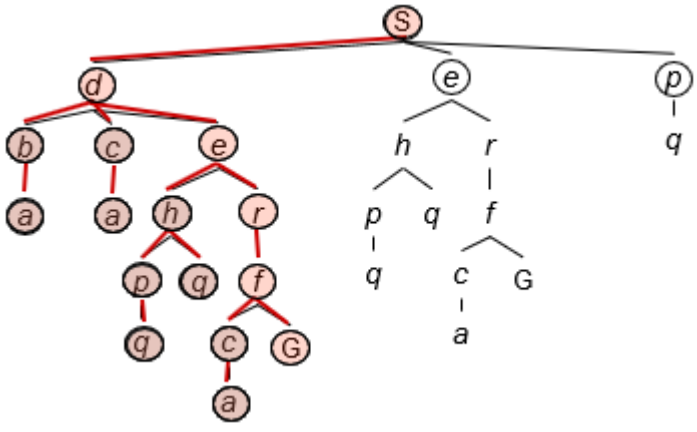


Źródło: materiały do wykładów CS188, UC Berkeley.

Depth-first search

Pierwsza strategia jest związana z głębokością budowanego drzewa. Pierwszy (najczęściej losowo wybrany) węzeł jest rozbudowaywany do samego dołu. Jest to możliwe dzięki implementacji tzw. stosu działającego na zasadzie LIFO (*last in, first out*) jako struktury danych dla *fringe*. Zatem ostatni sprawdzany węzeł będzie brany w pierwszej kolejności do dalszej eksploracji.

Grafika 3. Badanie przestrzeni za pomocą DFS. G = cel, S = początek.



Źródło: materiały do wykładów CS188, UC Berkeley.

Jeżeli chodzi o ocenę działania algorytmu, możemy stwierdzić, że:

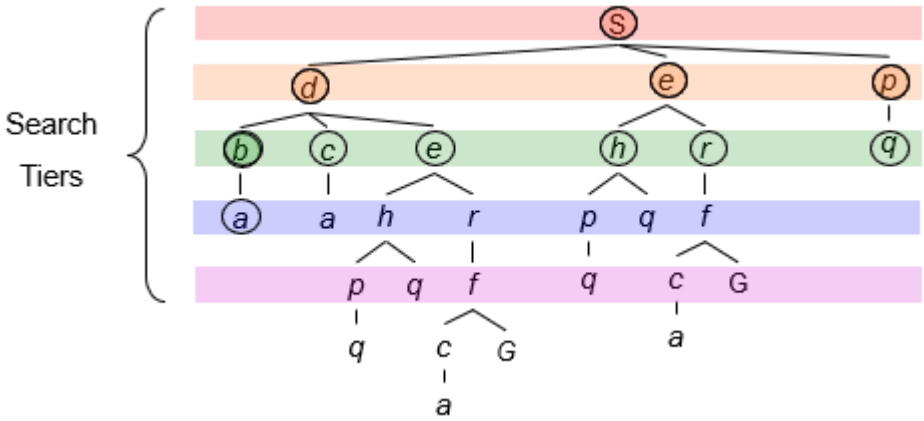
1. Nie ma gwarancji znalezienia rozwiązania w przypadku, kiedy algorytm wpadnie w nieskończoną pętlę. Konieczne jest wprowadzenie mechanizmu wychodzenia z tekiego cyklu.

2. Optymalne rozwiązanie może nie zostać znalezione ze względu na wcześniejsze znalezienie rozwiązania suboptymalnego.
3. Złożoność obliczeniowa jest na poziomie $O(b^m)$, gdzie *b* to jest liczba węzłów rosnąca wykładniczo w zależności od głębokości drzewa *m.* Rozwiązanie może się znajdować w ostatniej badanej ścieżce oraz na samym końcu drzewa.
4. Złożoność przestrzenna jest na poziomie $O(b * m)$, ponieważ rozważane są węzły dla każdego węzła *b* na każdym poziomie głębokości *m.*

Breadth-first search

Kolejna stategia od poprzedniej różni się strukturą danych wykorzystywaną do magazynowania węzłów we *fringe*. Opiera się ona na kolejkowaniu, czyli FIFO (*first in, first out*). Sprawia to, że węzły są brane do eksploracji warstwami – każdy węzeł jest losowo sprawdzany na aktualnym poziomie głębokości.

Grafika 4. Badanie przestrzeni za pomocą BFS.



Źródło: materiały do wykładów CS188, UC Berkeley.

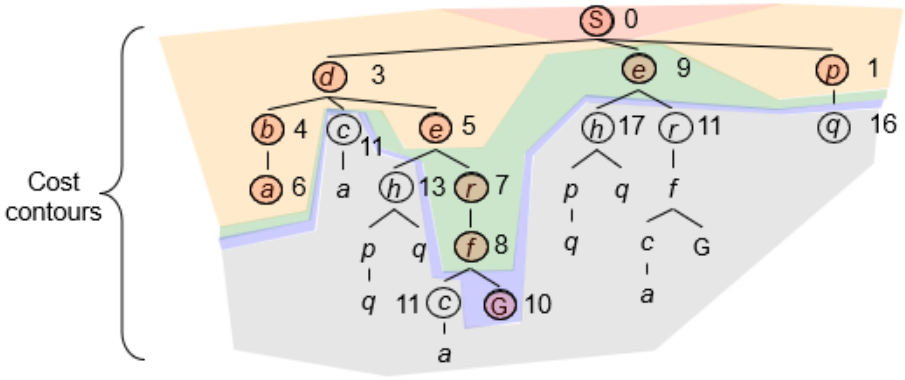
BFS cechuje się kolejno tym, że:

1. Gwarantuje znalezienie rozwiązania, ponieważ nie ma ryzyka wpadnięcia w pętlę, ponieważ liczba eksplorowanych poziomów drzewa *s* jest skończona.
2. Z perspektywy głębokości – jest optymalny. Natomiast warto wziąć pod uwagę fakt, że algorytm nie bierze pod uwagę kosztów ścieżek, zatem optymalność może być podważona.
3. Złożoność obliczeniowa jest na poziomie $O(b^s)$, analogicznie do DFS.
4. Złożoność przestrzenna jest taka sama jak obliczeniowa $O(b^s)$.

Uniform-cost search

UCS jest podobnym algorytmem do BFS, jednak uwzględnia kluczowy czynnik jakim jest koszt wykonania danej akcji. Strukturą danych wykorzystywaną w tym celu jest kolejkowanie z priorytetem, którym jest jak najniższy skumulowany koszt przejścia ścieżki.

Grafika 5. Badanie przestrzeni za pomocą UCS.



Źródło: materiały do wykładów CS188, UC Berkeley.

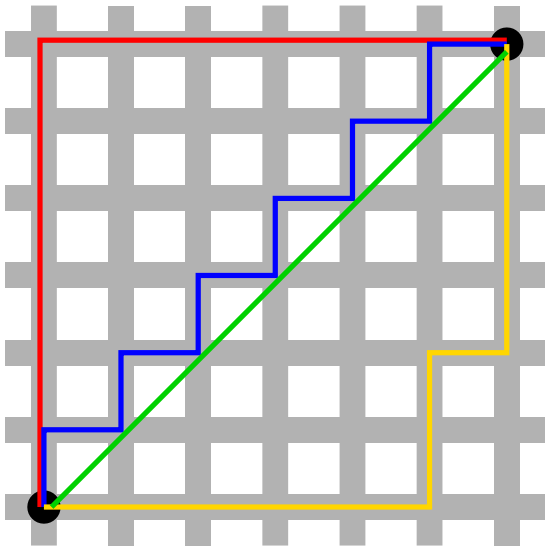
Jeżeli chodzi o efektywność UCS:

1. Gwarantuje znalezienie rozwiązania.
2. Gwarantuje znalezienie optymalnego rozwiązania w rozumieniu sumy kosztów dotarcia do celu.
3. Złożoność obliczeniowa jest na poziomie $O(b^{C/e})$, gdzie *C* to jest koszt optymalnej ścieżki prowadzącej do celu, natomiast wykonanie akcji kosztuje minimum *e.*
4. Złożoność przestrzenna działa na takiej samej zasadzie jak w BFS i wynosi tyle samo co obliczeniowa.

A* search

Jest algorytmem wykorzystującm informację zawartą w heurystyce, czyli funkcji estymującej jak blisko znajduje się cel. Heurystyka tworzona jest na potrzeby konkretnego problemu. Najprostrzymi są tzw. metryka miejska oraz odległość euklidesowa.

Grafika 6. Odległość euklidesowa (zielony) oraz metryki miejskie (pozostałe).



Źródło: Wikipedia.

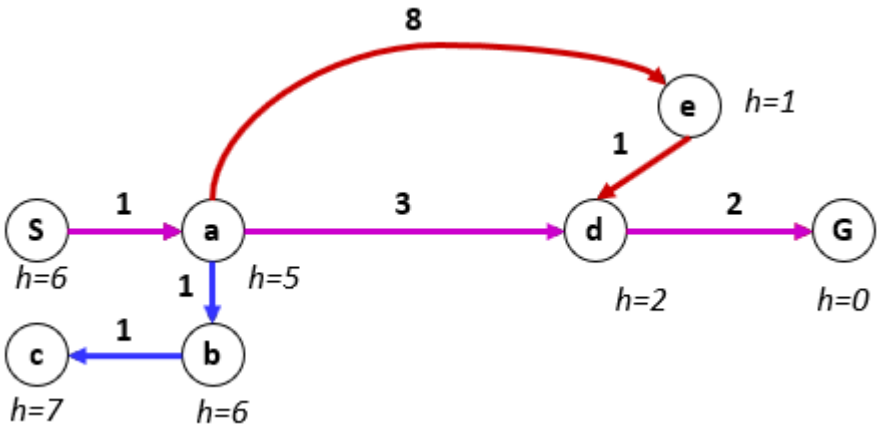
Aby heurystyka była efektywna, musi spełnić dwa łączące się warunki:

1. Musi być optymistyczna, czyli faktyczny koszt ścieżki musi być zawsze większy lub równy od kosztu szacowanego przez funkcję. W innym przypadku algorytm zatrzymywałby się na rozwiązaniach suboptymalnych. Jednak algorytm z użyciem pesymistycznej heurystyki będzie szybszy obliczeniowo.
2. Musi być stabilna, to znaczy że wartość heurystyki dla kązdego węzła powinna być mniejsza lub równa rzeczywistemu kosztowi węzłów.

Na potrzeby niniejszej pracy zaimplementowano funkcję opartą o faktyczne koszty ścieżki (estymacja = rzeczywistość). Wymagało to zwiększenia wykorzystania zasobów obliczeniowych ze względu na wykorzystanie BFS jako wsparcia dla A*. Polega ona na wielokrotnym obliczaniu wartości heurystyki dla głównego algorytmu, a następnie wybranie maksymalnej wartości z obliczonego zbioru (szczegóły w sekcji Kod).

A* jest połączeniem dwóch metod: **UCS g(n)** oraz ***greedy search* h(n)**, który opiera się tylko na informacji pochodzącej z heurystyki. Jest to **suma tych funkcji**. Algorytm zawsze znajduje optymalne rozwiązanie w zależności od zastosowanej heurystyki. Jego złożoność obliczeniowa również od niej zależy.

Grafika 7. Badanie przestrzeni za pomocą UCS, A oraz Greedy.*



Źródło: materiały do wykładów CS188, UC Berkeley.

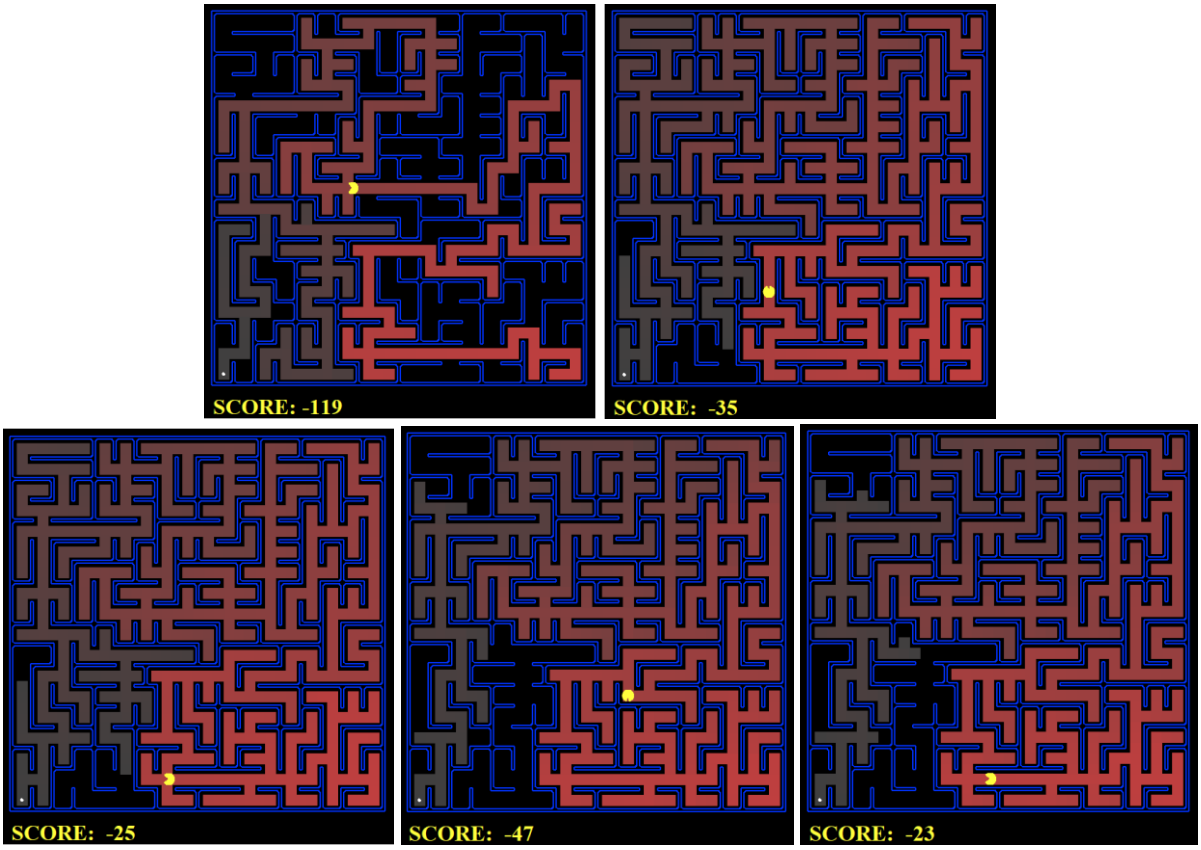
Działanie algorytmów

Jak wspominano we wstępie, algorytmy sprawdzono w na różnych problemach: klasyczny labirynt (*bigMaze*), otwarta przestrzeń (*openMaze*), oraz przestrzenie z wieloma punktami pożywienia, tj. na krańcach labiryntu oraz chaorycznie. Analizie poddano ilość eksplorowanych węzłów jak i całkowity koszt wybranej ścieżki. Każdy ruch (północ, południe, wschód, zachód) kosztuje Pacmana 1 punkt.

Klasyczny labirytnt

W tej klasie problemu Pacman musi znaleźć drogę do jednej kropki na mapie, do której prowadzi jedna ścieżka. Na grafikach widać obszary, które zostały sprawdzone. Im jaśniejsze i przechodzące w czerwony, tym wcześniej zostały zbadane, a pola całkowicie czarne w ogóle nie zostały odwiedzone. Na załączonych obrazkach widać wyraźnie podobieństwa i różnice w eksploracji między strategiami. Na pierwszy rzut oka najefektywniejszą strategią w tym przypadku jest DFS, ponieważ eksploracja ukierunkowana jest na rozwiązanie.

Grafika 8. Wizualizacja badania przestrzeni przez algorytmy kolejno: DFS, BFS, UCS, A (Manhattan), A* (Euclidean).*



Źródło: Opracowanie własne.

Powyższy wniosek potwierdza poniższa tabela. Ze względu na istnienie tylko jednego rozwiązania, koszt ścieżek dla wszystkich przypadków jest taki sam. Najlepszy rezultat osiągają kolejno: DFS, A* z użyciem metryki miejskiej oraz euklidesowej, BFS i UCS.

Tabela 1. Resultaty badania zamkniętej przestrzeni przez algorytmy kolejno: DFS, BFS, UCS, A (Manhattan), A* (Euclidean).*

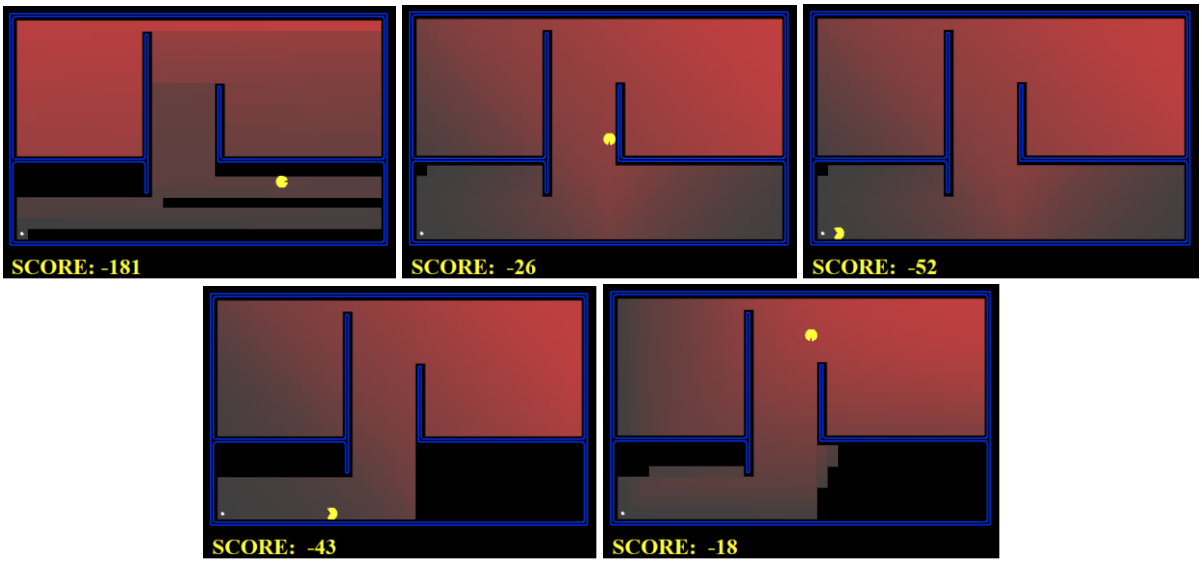
Labirynt klasyczny	***DFS***	***BFS***	***UCS***	***A* with Manhattan distance***	***A* with Euclidean distance***
Ilość węzłów	**390**	**620**	**620**	**549**	**557**
Koszt	**210**				

Źródło: Opracowanie własne.

Otwarty labirynt

Kolejnym wyzwaniem był labirynt o nieco innej strukturze, gdzie istnieje bardzo duża liczba możliwych rozwiązań z powodu istnienia otwartych przestrzeni. W tym przypadku koszt ścieżki odgrywa kluczową rolę. Wizualizacja wskazywałaby na najlepsze rozwiązanie zaproponowane przez A* z metryką miejską.

Grafika 9. Wizualizacja badania przestrzeni przez algorytmy kolejno: DFS, BFS, UCS, A (Manhattan), A* (Euclidean).*



Źródło: Opracowanie własne.

Wnioski wyciągnięte z grafik zostały potwierdzone liczbami zamieszczonymi w tabeli poniżej. A* zaoferował optymalne rozwiązanie przy najniższym wskaźniku zbadanych węzłów (kolejno 535 oraz 550) i kosztem równym 54. Ponadto stwierdzić można, że algorytm DFS najgorzej poradził sobie w tym środowisku znajdując rozwiązanie suboptymalne (koszt 298). BFS oraz UCS jednakowo mocno eksplorowały przestrzeń (zbadane węzły na poziomie 682) znajdując optymalne rozwiązanie.

Tabela 2. Resultaty badania otwartej przestrzeni przez algorytmy kolejno: DFS, BFS, UCS, A (Manhattan), A* (Euclidean).*

Labirynt otwarty	***DFS***	***BFS***	***UCS***	***A* with Manhattan distance***	***A* with Euclidean distance***
Ilość węzłów	**576**	**682**	**682**	**535**	**550**
Koszt	**298**	**54**			

Źródło: Opracowanie własne.

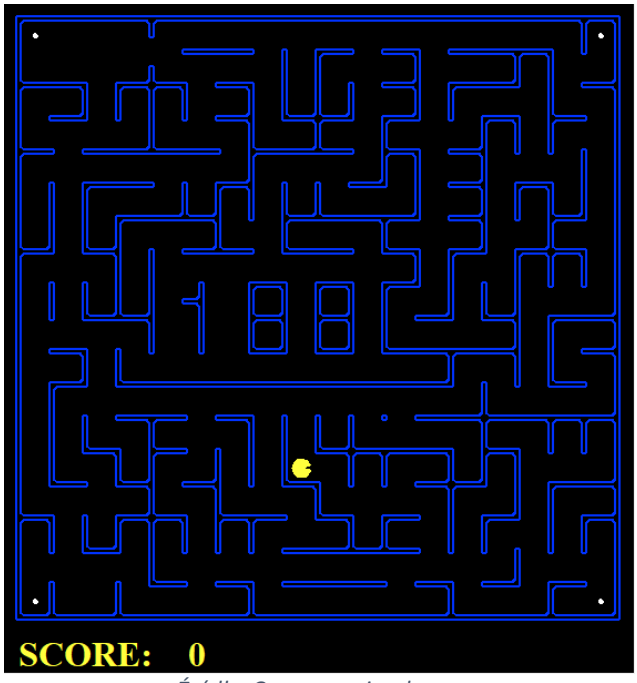
Labirynt z wieloma punktami

Czas obliczeń poprzednich problemów zamykał się w granicy do 0.1 sekundy. Od tego momentu złożoność obliczeniowa będzie większa, dlatego dodano kolejny miernik jakim jest czas wykonania.

Rozkład pożywienia na krańcach przestrzeni

Tym razem Pacman koniecznie musi odwiedzić wszystkie cztery rogi labiryntu. Do radzenia sobie z większą ilością punktów do odwiedzenia zastosowano heurestykę obliczaną sposobem opisanym we wstępie teoretycznym (tzw. *semi-lattice*).

Grafika 10. Wizualizacja problemu.



Źródło: Opracowanie własne.

Efektywność sprawdzanych algorytmów Prezentuje poniższa tabela. DFS mimo błyskawicznych obliczeń znalazł rozwiązanie suboptymalne. UCS o sekundę szybciej znalazł rozwiązanie względem BFS, natomiast obydwa zbadały taką samą przestrzeń i znalazły rozwiązanie optymalne. A* mimo bardzo dobrego rezultatu przestrzennego i znalezienia optymalnej ścieżki potrzebował prawie sześciokrotnie więcej czasu względem UCS.

Tabela 3. Rezultaty badania otwartej przestrzeni przez algorytmy kolejno: DFS, BFS, UCS, A.*

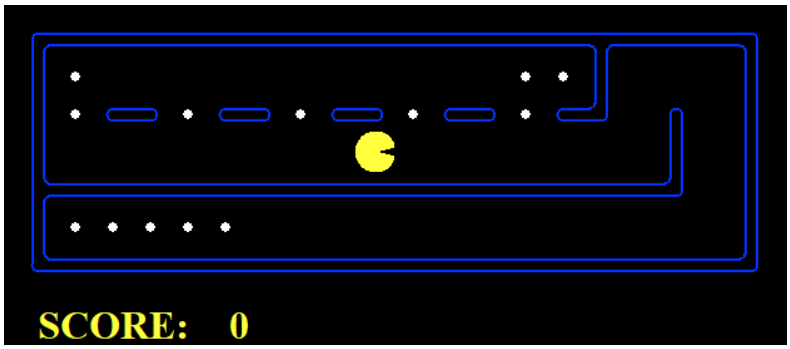
Krańce	***DFS***	***BFS***	***UCS***	***A****
Ilość węzłów	**509**	**7949**	**7949**	**2896**
Koszt	**302**	**162**	**162**	**162**
Czas (sekundy)	**0.9**	**13.8**	**12.7**	**72.1**

Źródło: Opracowanie własne.

Losowy rozkład pożywienia

Przedostatni podpunkt dotyczy trudniejszego układu labiryntu i pożywienia.

Grafika 11. Wizualizacja problemu.



Źródło: Opracowanie własne.

Poniższa tabela prezentuje wyniki symulacji. Rezultat jest zbliżony do poprzedniego problemu, gdzie DFS znalazł rozwiązanie suboptymalne, BFS oraz UCS praktycznie tak samo badały przestrzeń, natomiast A* osiągnął najlepsze wyniki kosztem czasu obliczeń.

Tabela 4. Rezultaty badania otwartej przestrzeni przez algorytmy kolejno: DFS, BFS, UCS, A.*

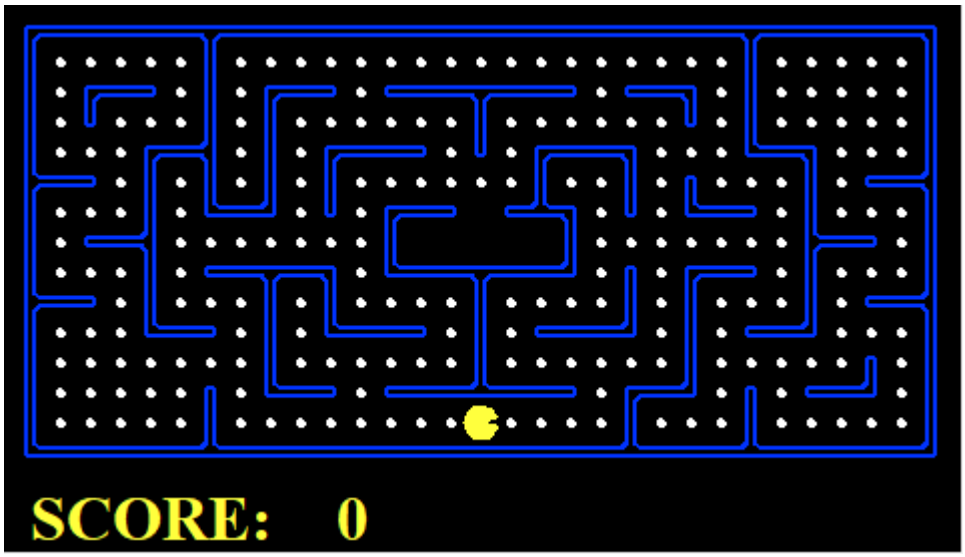
Krańce	***DFS***	***BFS***	***UCS***	***A****
Ilość węzłów	**361**	**16688**	**16688**	**4137**
Koszt	**216**	**60**	**60**	**60**
Czas (sekundy)	**0.1**	**3.5**	**3.7**	**30.9**

Źródło: Opracowanie własne.

Przestrzeń wypełniona pożywieniem

Ostatni podpunt ma na celu zaprezentowanie sytuacji, kiedy użycie efektywnego algorytmu z prawidłową heurystyką traci sens. Tak się dzieje kiedy problem jest bardzo wymagający obliczeniowo (patrz: obrazek poniżej). W takiej sytuacji celem jest znalezienie szybkiego rozwiązania suboptymalnego. W tym celu funkcja powinna poszukiwać ścieżki do kropki, która jest w danej chwili najbliżej Pacmana.

Grafika 12. Wizualizacja problemu.



Źródło: Opracowanie własne.

Z wyjątkiem DFS, pozostałe trzy algorytmy (BFS, UCS, A* z trywialną heurystyką) zwracają ścieżkę z kosztem na poziomie 350 ruchów i obliczeniach na poziomie 1 sekundy.

Podsumowanie

Na podstawie wyników zaprezentowanych symulacji można stwierdzić, że wybór algorytmu powinień być dokonany w zależności od problemu do rozwiązania oraz wymagań analityka. Przykładowo jeżeli zależy nam na znalezieniu jakiegokolwiek rozwiązania za pomocą szybkich obliczeń, pierwszym wyborem powinien być DFS. Jeżeli badana przestrzeń nie jest ograniczeniem, powinien zostać zastosowany BFS lub UCS w zależności od tego, czy koszt jest istotnym czynnikiem. Jeżeli ważna jest dokładność rozwiązania przy oszczędnej eksploracji przestrzeni, należy zastosować A* wraz z odpowiednio dobraną heurystyką.

Bibliografia

Pozycja główna:

Abbeel P., Klein D., i in. *CS188: Introduction to AI*. University of California. Berkeley. 2014. link: http://ai.berkeley.edu/home.html (dostęp: 10.12.2018).

Materiały do wykładów. http://ai.berkeley.edu/course_schedule.html (dostęp: 03.01.2019)

- Uninformed Search
- A* Search and Heuristics

Źródło danych (gra z funkcjami pomocniczymi):

- Project 1: Search, http://ai.berkeley.edu/project_log.html (dostęp: 03.01.2019)

Pozycje pomocnicze:

- wikipedia.org. (2019). Taxicab geometry. https://en.wikipedia.org/wiki/Taxicab_geometry [dostęp 27.01.2019].
- Medium.com (2019). AI — Teaching Pacman To Search With Depth First Search. https://medium.com/@lennyboyatzis/ai-teaching-pacman-to-search-with-depth-first-search-ee57daf889ab [dostęp 24.01.2019].
- Medium.com (2019). Solving the Traveling Pacman Problem – Robert Grosse – Medium. https://medium.com/@robertgrosse/solving-the-traveling-pacman-problem-39c0872428bc [dostęp 24.01.2019].
- theory.stanford.edu. (2019). Heuristics. http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html [dostęp 23.01.2019].