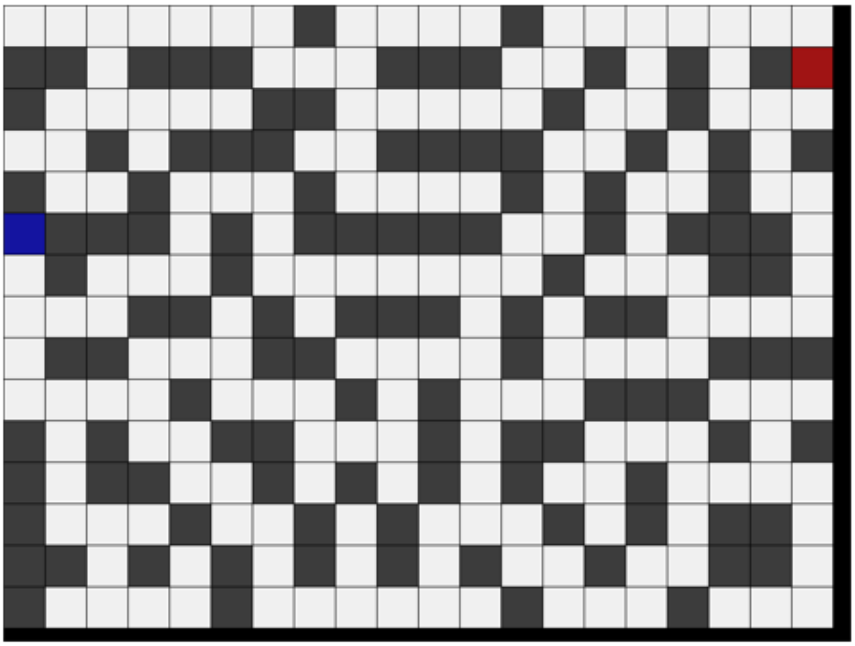
Sposoby znajdowania ścieżki w labiryncie w tym przykład zastosowania algorytmu A*

W zadaniu tym będziemy rozważać problem znajdowania ścieżki przez labirynt. Załóżmy, że mamy labirynt (jak na rysunku poniżej) z zaznaczonym punktem startu (pole niebieskie – wejście do labiryntu) oraz punktem końcowym (pole czerwone – wyjście z labiryntu tam chcemy dotrzeć). Poruszać można się po białych polach, czarne pola to ściany.



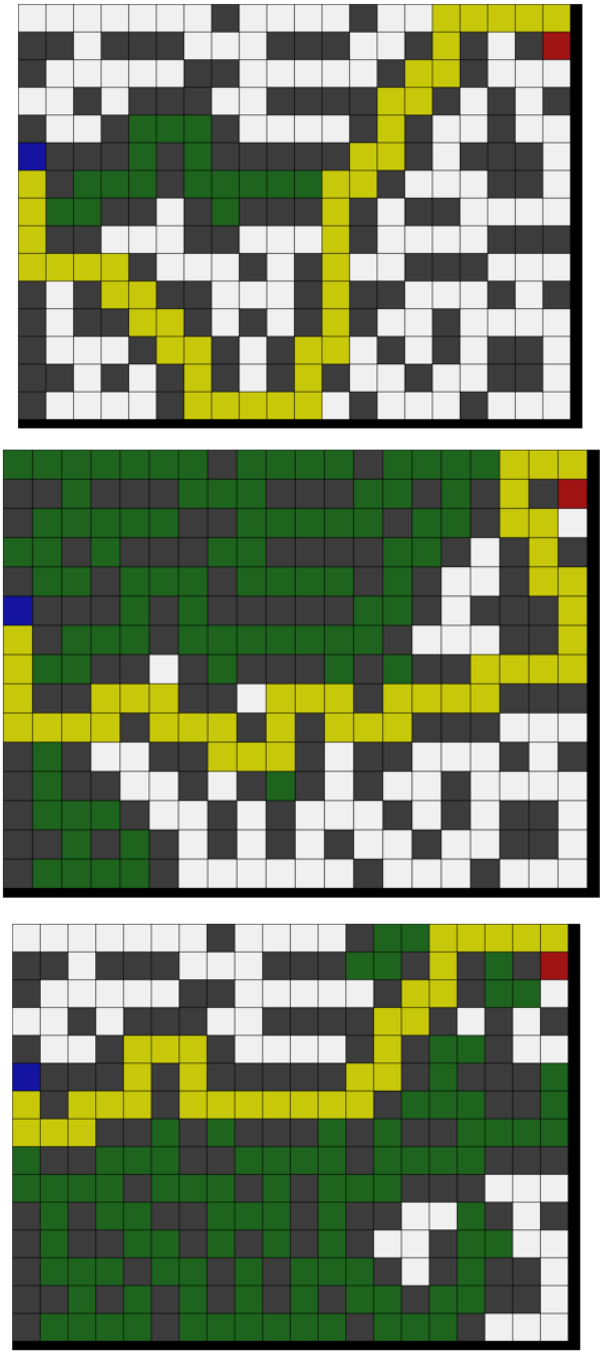
Na potrzeby ćwiczenia przyjmijmy, że labirynt zapisywany jest w pliku tekstowym (format csv tzn. liczby rozdzielone znakiem ;). 0 oznacza ścianę, 1 pole, gdzie można się poruszać, 2 pole startowe, 3 wyjście z labiryntu.

```
1;1;1;1;1;1;1;0;1;1;1;1;0;1;1;1;1;1;1;1
0;0;1;0;0;0;1;1;1;0;0;0;1;1;0;1;0;1;0;3
0;1;1;1;1;1;0;0;1;1;1;1;1;0;1;1;0;1;1;1
1;1;0;1;0;0;0;1;1;0;0;0;0;1;1;0;1;0;1;0
0;1;1;0;1;1;1;0;1;1;1;1;0;1;0;1;1;0;1;1
2;0;0;0;1;0;1;0;0;0;0;0;1;1;0;1;0;0;0;1
1;0;1;1;1;0;1;1;1;1;1;1;1;0;1;1;1;0;0;1
1;1;1;0;0;1;0;1;0;0;0;1;0;1;0;0;1;1;1;1
1;0;0;1;1;1;0;0;1;1;1;1;0;1;1;1;1;0;0;0
1;1;1;1;0;1;1;1;0;1;0;1;1;1;0;0;0;1;1;1
0;1;0;1;1;0;0;1;1;1;0;1;0;0;1;1;1;0;1;0
0;1;0;0;1;1;0;1;0;1;0;1;0;1;1;0;1;1;1;1
0;1;1;1;0;1;1;0;1;0;1;1;1;0;1;0;1;0;0;1
0;0;1;0;1;0;1;0;1;0;1;0;1;1;0;1;1;0;0;1
0;1;1;1;1;0;1;1;1;1;1;1;0;1;1;1;0;1;1;1
```

Generalnie istnieje wiele metod wyznaczania ścieżki, po której można dość od startu do pola wyjściowego. Problem można sprowadzić do odnajdywania ścieżki (np. najkrótszej) w grafie, zatem od biedy można go rozwiązać np. za pomocą algorytmu Dijkstry lub Floyda-Warshalla. My jednak z pewnych powodów (np. wydajnościowych) pomyślimy o nieco innych metodach. Swoją drogą to warto zastanowić się do czego praktycznie może służyć rozwiązywanie tego rodzaju problemów. Otóż czasem stosuje się tą metodę do symulacji „inteligencji" przeciwników w grach (przeciwnik dąży najkrótszą trasą do gracza lub innego ruchomego obiektu, pewnym -dość luźnym- przykładem może być tu PacMan choć tam zachowanie „duszków" było trochę bardziej skomplikowane niż proste ściganie gracza)

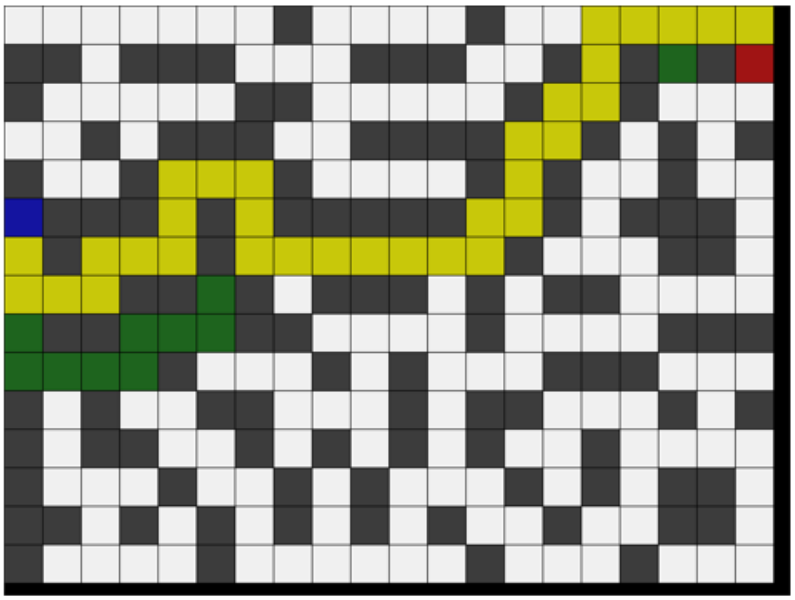
Przy naszym problemie trzeba zauważyć dwie kwestie:

1) Może istnieć kilka dróg prowadzących od startu do mety np. poniżej (drogę zaznaczyłem na żółto, to jest ten sam labirynt co na pierwszym rysunku)



Oczywiście optymalna jest ta droga, która jest najkrótsza 😊 Na przykładach powyżej optymalna ścieżka jest na ostatnim (najniższym) rysunku.

2) Drugą z kwestii, na którą warto zwrócić uwagę jest szybkość działania. Wiąże się ona z ilością pól, jaką trzeba było przeanalizować, aby znaleźć jak najlepszą drogę (a nawet drogę „globalnie" optymalną). Pola, jakie użyty przeze mnie algorytm analizował (oczywiście oprócz samej żółtej ścieżki łączącej start z metą, oznaczałem na zielono). Jak użyje się odpowiedniego algorytmu optymalnej ścieżki można szukać analizując bardzo mało pól, dla naszego przykładu wygląda to tak jak poniżej (tym razem zielonych pól jest dużo mniej niż na ostatnim przykładzie – z punktu 1, który też wyznaczył optymalną ścieżkę) :



Proszę też popatrzyć na przykład 1 ze strony drugiej. Tam też odwiedzono mało pól (zielonych) ale wyznaczona ścieżka – choć prowadzi do wyjścia- nie jest optymalna.

Przegląd wybranych algorytmów do znajdowania dróg w labiryncie.

Dwa najprostsze algorytmy do wyznaczania ścieżki w labiryncie oparte, czy też inspirowane, są przez algorytmy przeszukiwania grafu w głąb (DFS) i w szerz (BFS).

DFS

Zacznijmy rozważania od algorytmu opartego na DFS (dla uproszczenia będziemy go tu nazywać algorytmem DFS). Mój opis będzie miał formę skrótową, przedstawiającą ogólną ideę. W algorytmie tym musimy utworzyć tablicę, w której zaznaczamy które pola odwiedziliśmy oraz strukturę danych typu stos. Dodatkowo, -dla odtworzenia ścieżki- potrzebujemy tablicy, która będzie nam trzymać „poprzedniki". Algorytm jest prosty.

1) Dodajemy pole startowe do stosu.
2) Dopóki stos nie jest pusty (lub nie nastąpi przerwania na skutek dotarcia do mety) robimy co następuje (pętla while):
   a) Zdejmujemy element (pole) ze stosu, nazwijmy go polem „aktualnym". Zaznaczamy, że pole „aktualne" zostało odwiedzone.

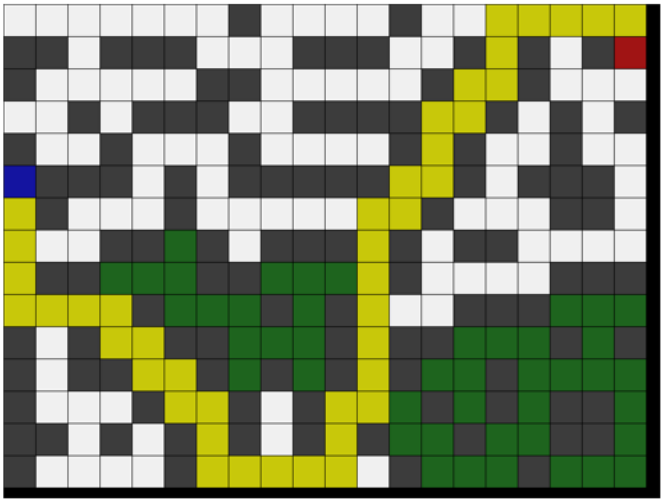
b) Sprawdzamy czy pole „aktualne” ma nieodwiedzonych sąsiadów, jeśli tak, wszystkich z nich dodajemy do stosu. W tablicy „poprzedników” jako poprzedników tych pól ustawiamy pole „aktualne”. Sąsiadów do stosu najlepiej dodawać w sposób losowy (oczywiście dodajemy ich do „góry” stosu, chodzi o to, by -jeśli aktualne pole ma kilku nieodwiedzonych sąsiadów- dodawać ich w losowej kolejności a nie wg jakiegoś schematu. Jeśli jeden z sąsiadów jest polem finalnym (meta lub stop) to przerywamy algorytm – ważne by zrobić to po wpisaniu poprzedników sąsiadów do tablicy.

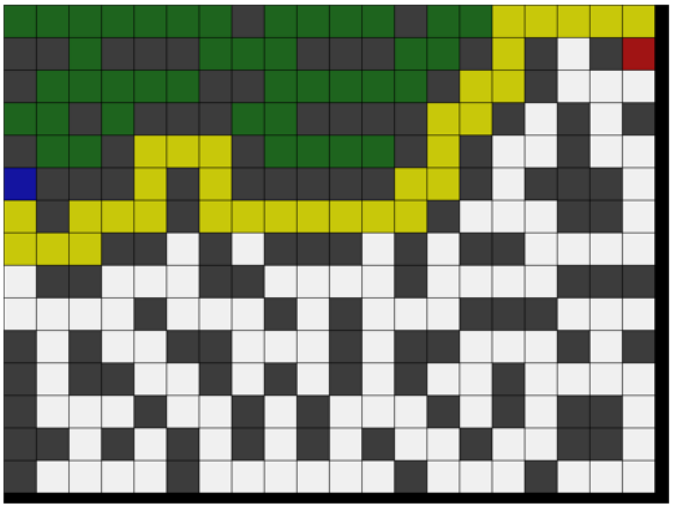
c) Przechodzimy do kolejnej iteracji pętli while.

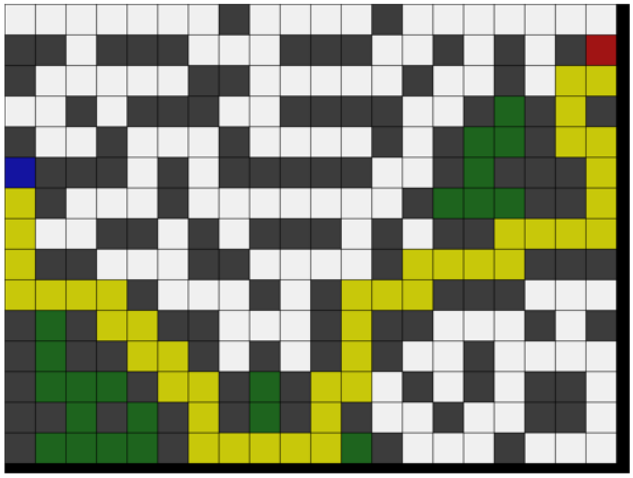
3) Po zakończeniu algorytmu aby odtworzyć ścieżkę sprawdzając od „mety” z tabelki „poprzedników” odczytujemy poprzedników aż dojdziemy do startu.

Uwaga. Jeśli istnieje ścieżka łącząca start z metą to algorytm ją znajdzie (i nastąpi jego przerwanie). Algorytm ten jest w stanie generować różne drogi (na skutek losowej kolejności dodawania sąsiadów aktualnego wierzchołka). Może się zdarzyć, że wyznaczy on drogę optymalną, może się też zdarzyć, że zrobi to bardzo dobrym „kosztem” (tzn. analizując mało zbędnych pól), jednakże jest to raczej rzadka sytuacja.

Poniżej kilka wyników DFS:

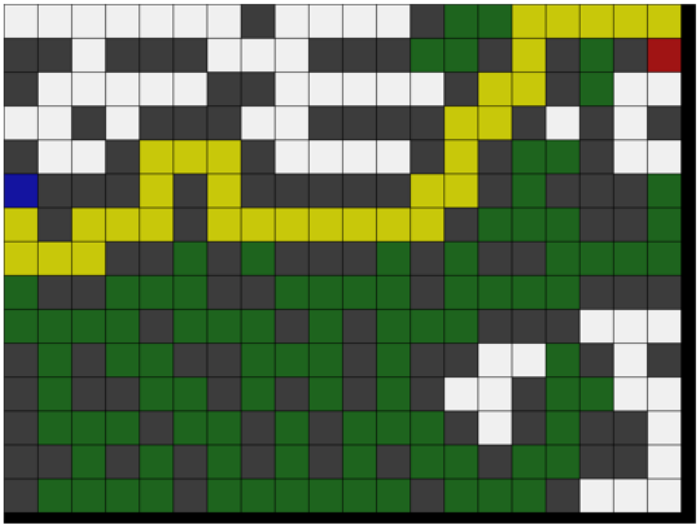


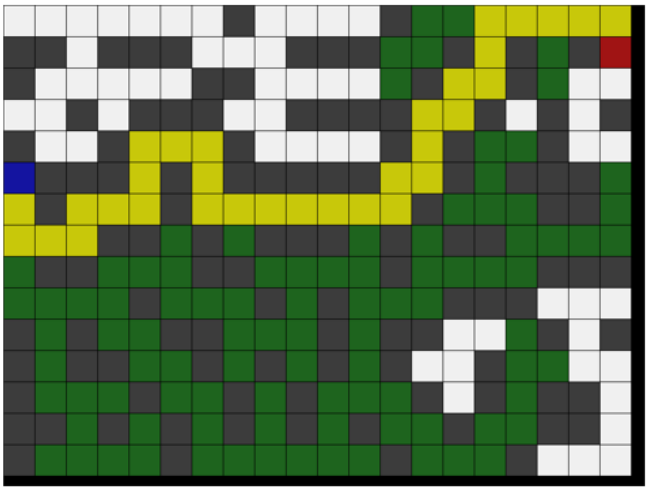




BFS

Algorytm BFS działa DOKŁADNIE tak samo jak DFS, z tym, że elementy składamy nie na stosie (LIFO) tylko w kolejce (FIFO). Dlatego też nie umieszczę tu szczegółowego opisu (jest taki sam jak dla DFS). Istotne jest to, że dla tego problemu algorytm BFS ZAWSZE znajduje rozwiązanie optymalne (najkrótsza ścieżka łącząca start z metą). Niestety, jest to bardzo często okupione koniecznością analizy wielu zbędnych pól (z reguły jest ich dużo więcej niż dla algorytmu DFS). Co istotne (w przeciwieństwie do DFS) wyniki BFS są w miarę powtarzalne (choć czasem mogą się pojawić pewne bardzo drobne różnice związane z tym, które pole jest dodawane jako pierwsze)





Podsumowując BFS znajdzie nam optymalną ścieżkę, ale w dość nieoptymalny sposób. DFS znajdzie nam jakąś ścieżkę, często analizując znacznie mniej pól niż BFS (choć ściśle rzecz biorąc to ostatnie zależy od rodzaju labiryntu)

Prosty algorytm zachłanny

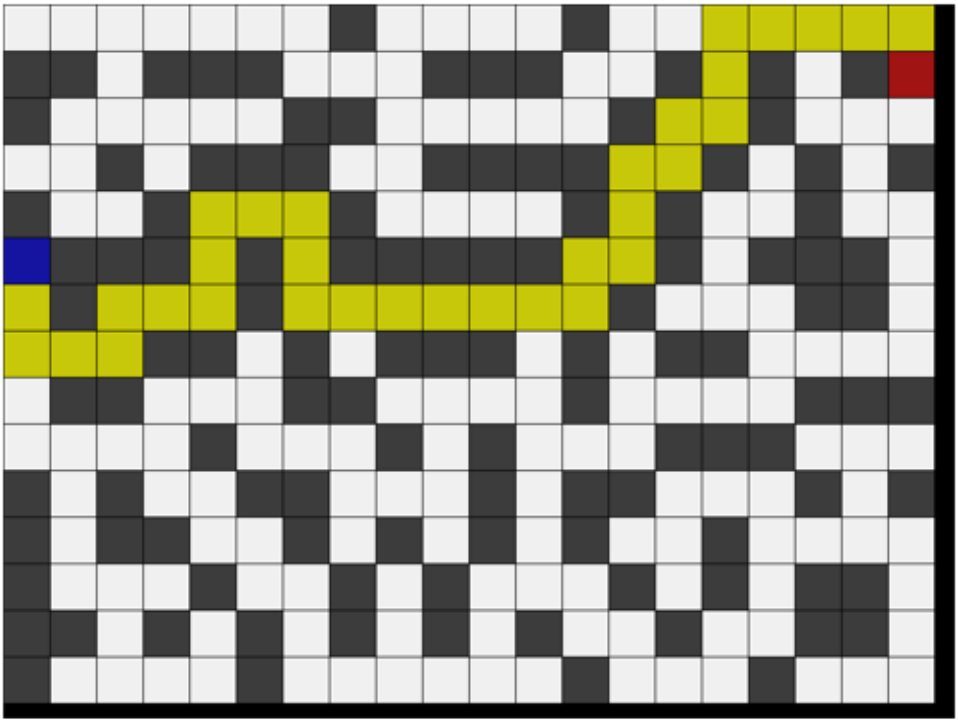
Zastanówmy się, czy można spróbować opracować inny algorytm, który wyznaczałby nam (najlepiej optymalną) ścieżkę, analizując stosunkowo mało pól. Tu nasuwa się na myśl pewien prosty algorytm zachłanny.
Algorytm zachłanny to taki algorytm, który w każdym kroku dokonuje zachłannego (tzn. w danym momencie najlepszego rozwiązania częściowego).

W naszym problemie algorytm ten działałby tak jak DFS lub BFS ale elementy trzymałby nie w stosie lub zwykłej kolejce a w kolejce priorytetowej. Priorytetem kolejki byłaby mała odległość od mety (mierzona np. jako metryka „taksówkowa” inaczej zwana metryką „Manhattan”).
Najpierw wyznaczmy sobie (dla każdego pola labiryntu także dla ścian) odległości od mety (dla ułatwienia liczyłem ją sobie raz i trzymałem w osobnej tablicy):

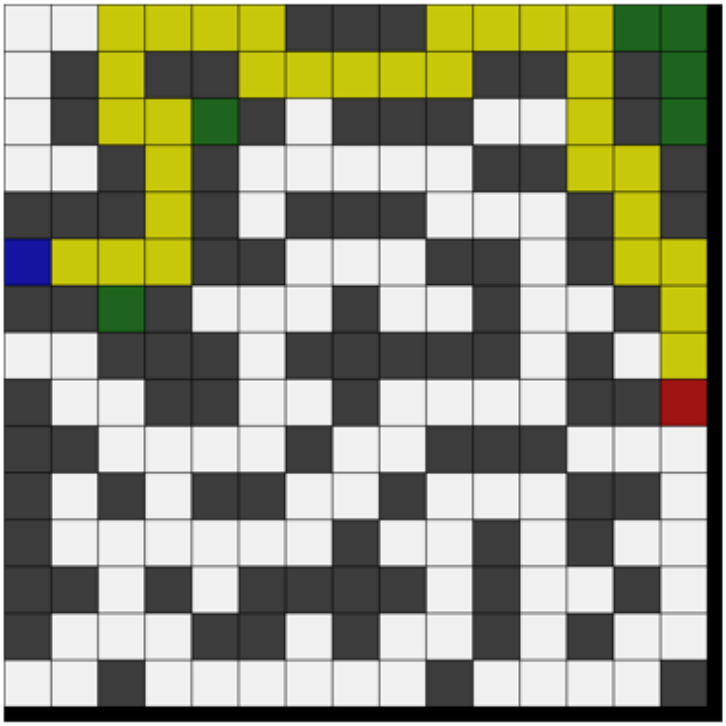
20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2
22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4
24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5
25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6
26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7
27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10
30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12
32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13

PS: Te wyznaczone tu odległości metodą metryki taksówkowej wyznaczone są metodą heurystyczną, zatem sam opisywany tu algorytm (podobnie jak algorytm A*) jest algorytmem heurystycznym 😊

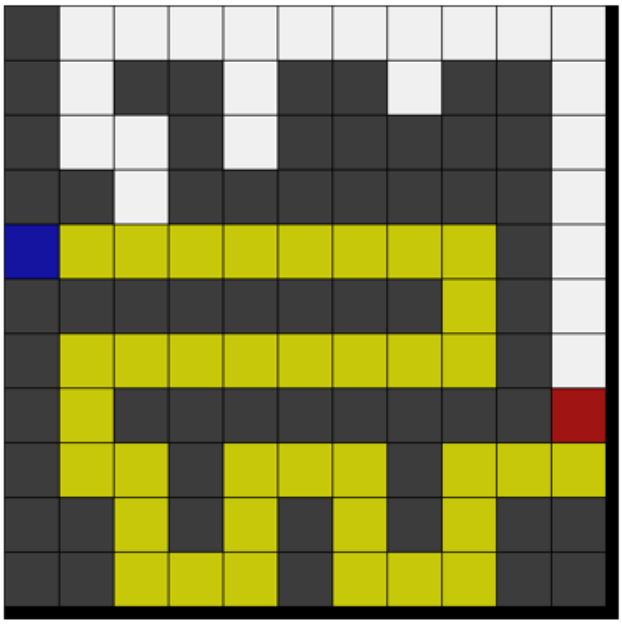
Akurat (przypadkowo) dla naszego labiryntu analizowanego od początku tutoriala algorytm od razu wyznacza optymalną ścieżkę w najbardziej optymalny sposób 😊



Ale nie zawsze jest tak idealnie. Czasami (bardzo często) algorytm analizuje jakieś dodatkowe pola np. tutaj:

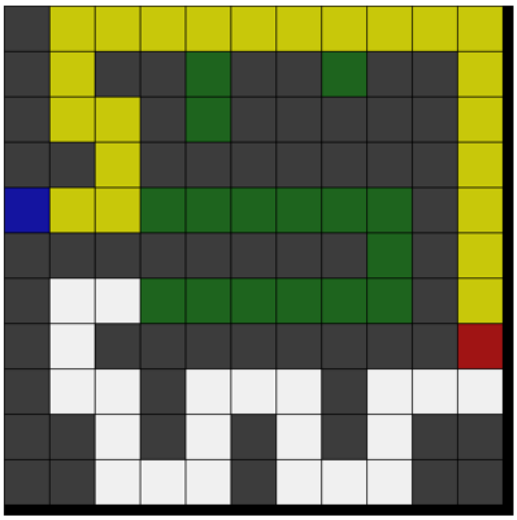


Co gorsza czasami (dla bardzo „złośliwych” labiryntów) zwraca on nieoptymalny wynik (tzn. ścieżkę, która nie jest najkrótsza):

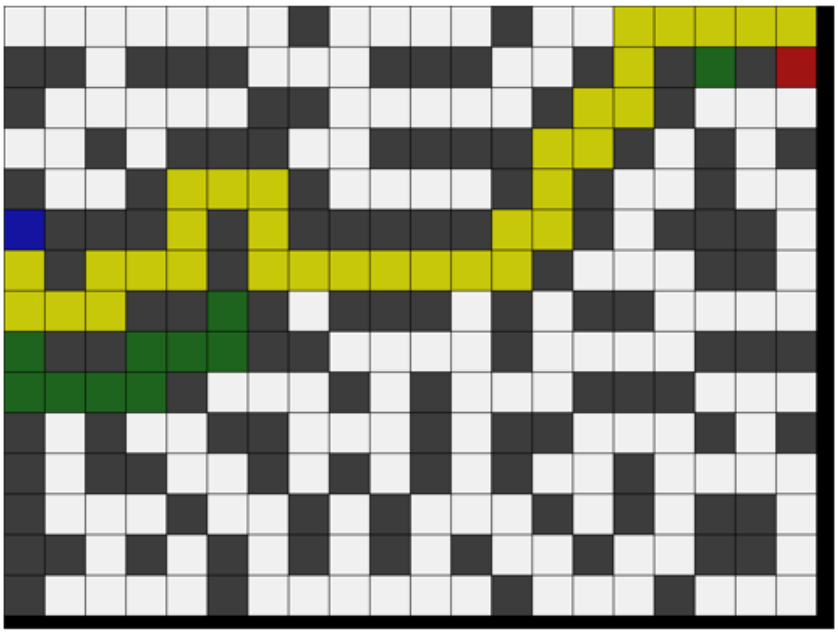


Algorytm A *

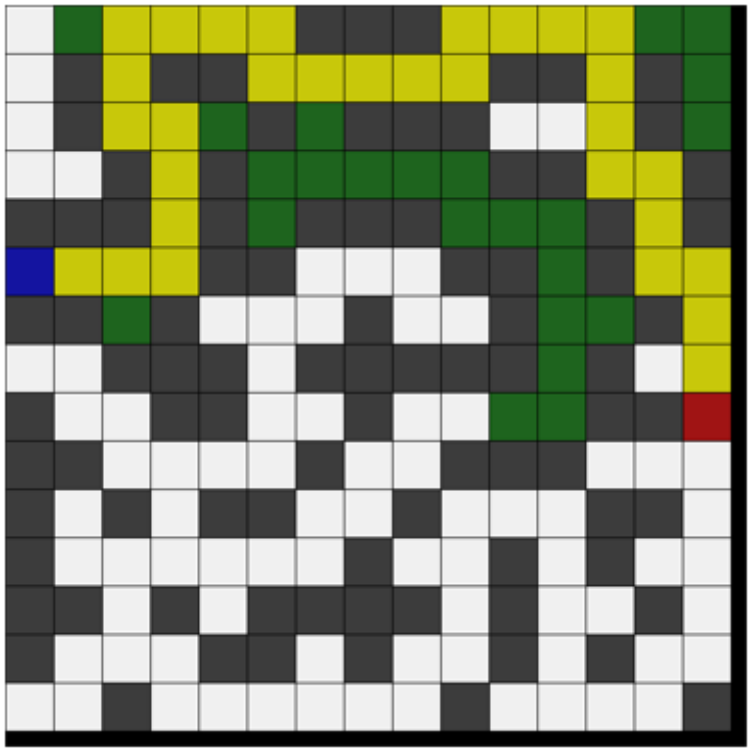
Zwracanie nieoptymalnej ścieżki to duży minus algorytmu ale można temu zaradzić stosując tzw. algorytm A *. W algorytmie tym wagą jest nie sama odległość od mety ale odległość od mety zsumowana z liczbą kroków, jakie trzeba przebyć by osiągnąć aktualne pole (chodzi o liczbę długość ścieżki od startu do aktualnego pola, którą trzeba sobie trzymać w osobnej tablicy). Wtedy nasz algorytm daje taki wynik dla problematycznego labiryntu:



Oraz dla naszego testowego:



No i dla labiryntu pokazanego u dołu 7 strony tutoriala:



Jeśli chcą Państwo dowiedzieć się więcej w tym temacie to polecam fajny wykład z Harwardu, który jest dostępny tutaj:
https://cs50.harvard.edu/ai/2020/weeks/0/ (o rozwiązywaniu labiryntów zaczyna się w 30 minucie 40 sekundzie filmu). Przyznam szczerze, że przygotowując ten tutorial trochę się nim zainspirowałem 😊