ВАРНЕНСКИ СВОБОДЕН УНИВЕРСИТЕТ
„ЧЕРНОРИЗЕЦ ХРАБЪР”

ФАКУЛТЕТ „СОЦИАЛНИ, СТОПАНСКИ И КОМПЮТЪРНИ НАУКИ”

КАТЕДРА „КОМПЮТЪРНИ НАУКИ”

КУРСОВА РАБОТА

по дисциплина: Програмиране и Алгоритми

на тема:

Алгоритми за намиране на най-кратък път в граф

Изготвил:
Христо Николаев Иванов
фак. № 257330006, курс първи
Специалност: Софтуерно инженерство и мениджмънт
Форма на обучение: дистанционно

Проверил:
Научна степен, име и фамилия
/...../

Варна, 2026 г.

СЪДЪРЖАНИЕ

Въведение

Алгоритмите за намиране на най-кратък път са едни от най-фундаменталните концепции в теорията на графите и компютърните науки. Те намират приложение в множество области – от GPS навигация и мрежови протоколи до изкуствен интелект и компютърни игри.

Настоящият курсов проект представя практическа имплементация на три класически алгоритъма за намиране на най-кратък път: алгоритъма на Дийкстра, алгоритъма A* и алгоритъма на Белман-Форд. За демонстрация на тяхната работа е използвана играта **PetWars** – походова стратегическа игра, вдъхновена от класиката Heroes of Might and Magic, в която котки се сражават срещу кучета за териториално надмощие.

В играта PetWars алгоритмите се използват за автоматично намиране на оптимален път от текущата позиция на героя до избрано поле на картата. Специално разработеният за тази курсова работа демо режим позволява визуализация на работата на алгоритмите стъпка по стъпка, което направи проекта подходящ и за образователни цели.

Фигура 1: Начален екран на играта PetWars

1. Теоретична част

1.1 Представяне на избраните алгоритми

В проекта са имплементирани три алгоритъма за намиране на най-кратък път в граф:

- Алгоритъм на Дийкстра – класически алгоритъм за графи с неотрицателни тегла
- Алгоритъм A* – разширение на Дийкстра с евристична функция
- Алгоритъм на Белман-Форд – универсален алгоритъм, работещ и с отрицателни тегла

Тези алгоритми са фундаментални в теорията на графите и намират широко приложение в компютърните игри, навигационните системи и мрежовите протоколи.

1.2 Алгоритъм на Дийкстра

Алгоритъмът на Дийкстра е класически алгоритъм за намиране на най-кратък път от един източник до всички останали върхове в граф с неотрицателни тегла на ребрата. Разработен е от холандския учен Едсхер Дийкстра през 1956 г.

Основна идея и принцип на работа:

- Поддържа се множество от посетени върхове и приоритетна опашка с непосетени върхове
- За всеки връх се пази текущото най-кратко разстояние от началния връх
- На всяка стъпка се избира непосетеният връх с най-малко разстояние
- Актуализират се разстоянията до съседите на избрания връх

Дефиниции и математически модел:

Нека $G = (V, E)$ е граф с множество от върхове V и множество от ребра E. За всяко ребро $(u, v) \in E$ има тегло $w(u, v) \geq 0$. Алгоритъмът намира $d[v]$ – минималното разстояние от началния връх s до всеки връх $v \in V$.

Приложения:

- GPS навигация и маршрутизиране
- Мрежови протоколи (OSPF)
- Компютърни игри (AI pathfinding)

Сложност: O((V+E) log V), където V е броят на върховете, а E е броят на ребрата.

1.3 Алгоритъм A*

Алгоритъмът A* е разширение на алгоритъма на Дийкстра, което използва евристична функция за насочване на търсенето към целта. Това го прави значително по-ефективен в практически приложения.

Основна идея и принцип на работа:

Използва функция f(n) = g(n) + h(n), където:

- g(n) е реалната цена от началото до текущия връх
- h(n) е евристичната оценка от текущия връх до целта
- В проекта се използва Manhattan distance като евристика: $h(n) = |x_1 - x_2| + |y_1 - y_2|$
- Алгоритъмът приоритизира върхове, които са по-близо до целта

Дефиниции и математически модел:

Евристичната функция h(n) трябва да бъде допустима (admissible), т.е. никога да не надценява реалното разстояние до целта. Manhattan distance е допустима евристика за движение в 4 посоки.

Приложения:

- Компютърни игри (основен алгоритъм за AI pathfinding)
- Роботика и автономни превозни средства
- Планиране на задачи

Сложност: O((V+E) log V) в най-лошия случай, но практически много по-бърз от Дийкстра.

1.4 Алгоритъм на Белман-Форд

Алгоритъмът на Белман-Форд намира най-кратките пътища от един източник до всички върхове, като може да работи и с отрицателни тегла на ребрата (за разлика от Дийкстра).

Основна идея и принцип на работа:

- Извършва V-1 итерации, където V е броят на върховете
- На всяка итерация "релаксира" всички ребра в графа
- Релаксация означава проверка дали пътят през дадено ребро е по-кратък

Дефиниции и математически модел:

Релаксация на ребро (u, v): ако d[u] + w(u, v) < d[v], тогава d[v] = d[u] + w(u, v). След V-1 итерации, ако все още има ребра за релаксация, графът съдържа отрицателен цикъл.

Приложения:

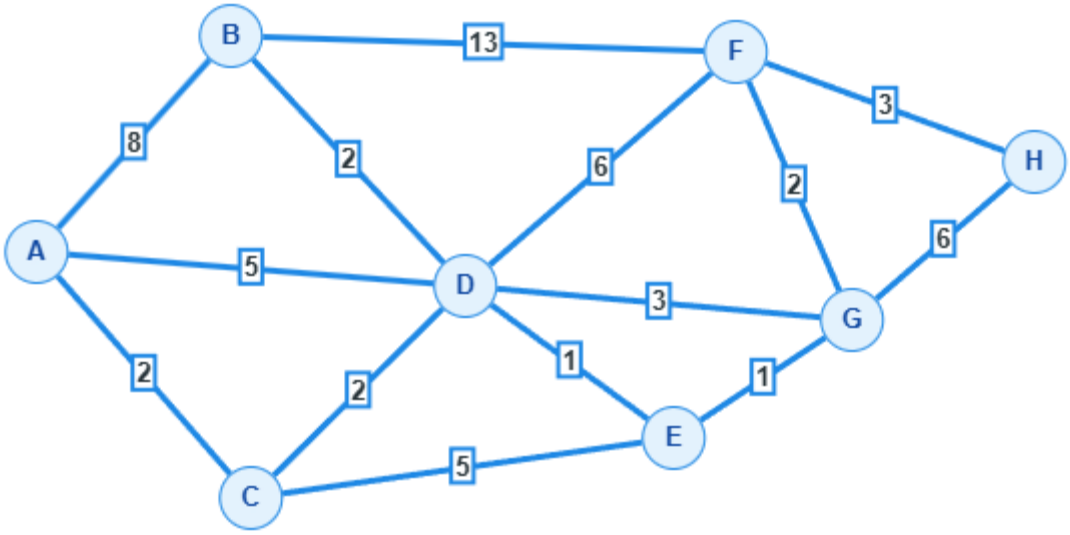
- Мрежови протоколи (RIP, BGP)
- Откриване на отрицателни цикли
- Финансов арбитраж

Сложност: O(V*E), което е по-бавно от Дийкстра, но по-универсално.

1.5 Сравнение на алгоритмите

Алгоритъм	**Сложност**	**Предимства**	**Недостатъци**
Дийкстра	O((V+E) log V)	Оптимален, надежден	Изследва всички посоки
A*	O((V+E) log V)	Насочено търсене, бърз	Изисква добра евристика
Белман-Форд	O(V*E)	Работи с отрицателни тегла	По-бавен

1.6 Поглед отблизо към Дийкстра с примерен граф:



Фигура 2: Примерен граф за демонстрация на алгоритъма на Дийкстра

V	A	B	C	D	E	F	G	H
A	0A	8A	2A	5A	∞	∞	∞	∞
C		8A	2A	4C	7C	∞	∞	∞
D		6D		4C	5D	10D	7D	∞
E		6D			5D	10D	6E	∞
B		6D				10D	6E	∞
G						8G	6E	12G
F						8G		11F
H								11F

Описание как работи алгоритъма стъпка по стъпка

Алгоритъмът на Дийкстра работи по следния начин:

1. Започваме от началния връх A и задаваме разстоянието до него равно на 0, а до всички останали върхове - безкрайност.
2. На всяка итерация избираме непосетения връх с най-малко текущо разстояние и го "заключваме" - това разстояние е окончателно.
3. След заключване актуализираме разстоянията до всички съседи на текущия връх: ако пътят през него е по-кратък от досегашния, записваме новата стойност.
4. Повтаряме стъпки 2-3 докато достигнем целевия връх или обходим всички върхове.

Индексът до всяко число в таблицата показва през кой връх минава най-краткият път до момента.

Начален връх: A, Целеви връх: H

Числата в таблицата представляват сумарното разстояние от A до съответния връх.

Итерация 1: Посещаваме A

Разстояние до A = 0

Изчисляваме сумарните разстояния до съседите: B: 0 + 8 = 8A C: 0 + 2 = 2A D: 0 + 5 = 5A E, F, G, H = безкрайност

Заключваме A = 0

Следващ: C (най-малко = 2)

Итерация 2: Посещаваме C

Разстояние до C = 2

Актуализираме сумарните разстояния: D: min(5, 2+2) = 4C (по-кратък път през C) E: 2 + 5 = 7C

Заключваме C = 2

Следващ: D (= 4)

Итерация 3: Посещаваме D

Разстояние до D = 4

Актуализираме сумарните разстояния: B: min(8, 4+2) = 6D E: min(7, 4+1) = 5D F: 4 + 6 = 10D G: 4 + 3 = 7D

Заключваме D = 4

Следващ: E (= 5)

Итерация 4: Посещаваме E

Разстояние до E = 5

Актуализираме сумарните разстояния: G: min(7, 5+1) = 6E

Заключваме E = 5

Следващ: B (= 6, избираме B преди G)

Итерация 5: Посещаваме B

Разстояние до B = 6

Актуализираме сумарните разстояния: F: min(10, 6+13) = 10D (без промяна, 19 > 10)

Заключваме B = 6

Следващ: G (= 6)

Итерация 6: Посещаваме G

Разстояние до G = 6

Актуализираме сумарните разстояния: F: min(10, 6+2) = 8G (G-F = 2) H: 6 + 6 = 12G

Заключваме G = 6

Следващ: F (= 8)

Итерация 7: Посещаваме F

Разстояние до F = 8

Актуализираме сумарните разстояния: H: min(12, 8+3) = 11F

Заключваме F = 8

Следващ: H (= 11)

Итерация 8: Посещаваме H

Разстояние до H = 11

Няма непосетени съседи

Заключваме H = 11

КРАЙ

Резултат

Най-кратък път от A до H = 11

Път: A, C, D, E, G, F, H

2. Пример за работата на алгоритмите

В играта PetWars алгоритмите се използват за намиране на път от текущата позиция на героя до избрана целева клетка на картата. Картата е представена като двумерна решетка (14x10 клетки), където всяка клетка може да бъде проходима (стойност 1) или непроходима (стойност 0).

Стъпки на изпълнение (на примера на Дийкстра):

1. Инициализация: началната позиция получава разстояние 0, всички останали - безкрайност

2. Добавяне на началната позиция в приоритетната опашка

3. Извличане на върха с най-малко разстояние от опашката

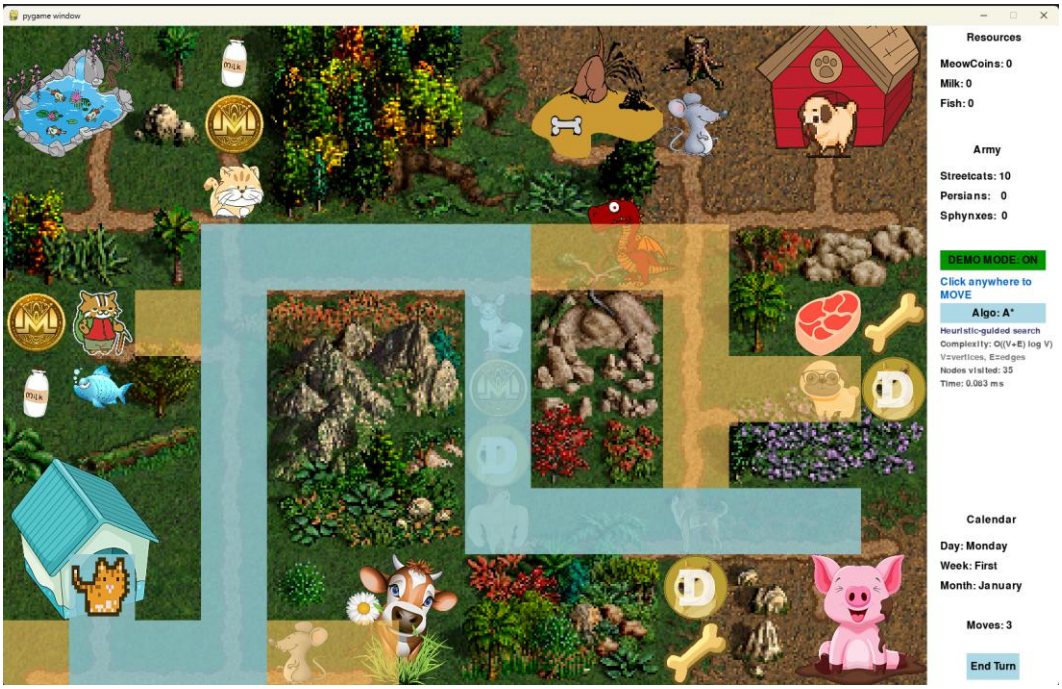
4. За всеки съсед (4 посоки: горе, долу, ляво, дясно): ако новият път е по-кратък, актуализиране на разстоянието

5. Повтаряне докато се достигне целта или опашката се изпразни

6. Възстановяване на пътя чрез обратно проследяване от целта към началото

В демо режима на играта визуализацията показва:

- Оранжеви клетки – посетени върхове (вече изследвани)
- Жълти клетки – върхове в опашката (frontier)
- Червена клетка – текущо разглеждан връх
- Сини клетки – намереният най-кратък път



Фигура 3: Визуализация на алгоритъма в демо режим

3. Програмна реализация - https://github.com/hristogwivanov/PetWars

Проектът е реализиран на Python с използване на библиотеката Pygame за графичния интерфейс. Основните файлове са:

- main.py – главен файл с игровия цикъл и визуализация
- pathfinding.py – имплементация на трите алгоритъма
- gamedata.py – класове за герои, ресурси и AI противник
- constants.py – константи, цветове и карти на терена
- interface.py – UI компоненти (бутони, брояч на ходове)

3.1 Имплементация на алгоритъма на Дийкстра

```
def dijkstra_path(start, goal, terrain_map):
    """
    Find shortest path using Dijkstra's algorithm.

    Args:
        start: Tuple (x, y) - starting position
        goal: Tuple (x, y) - destination position
        terrain_map: 2D list where 1 = walkable, 0 = blocked

    Returns:
```

```
    List of (x, y) tuples representing the path
"""
import heapq

# Priority queue: (distance, (x, y))
open_set = [(0, start)]
g_score = {start: 0}
came_from = {}

while open_set:
    # Get node with smallest distance
    current_dist, current = heapq.heappop(open_set)

    # Goal reached - reconstruct path
    if current == goal:
        path = []
        while current in came_from:
            path.append(current)
            current = came_from[current]
        path.append(start)
        return path[::-1]

    if current_dist > g_score.get(current, float('inf')):
        continue

    x, y = current
    # Explore 4-directional neighbors
    for dx, dy in [(0, -1), (0, 1), (-1, 0), (1, 0)]:
        nx, ny = x + dx, y + dy

        if not (0 <= nx < MAP_WIDTH and 0 <= ny < MAP_HEIGHT):
            continue
        if terrain_map[ny][nx] != 1:
            continue

        tentative_g = g_score[current] + 1

        if tentative_g < g_score.get((nx, ny), float('inf')):
            came_from[(nx, ny)] = current
            g_score[(nx, ny)] = tentative_g
            heapq.heappush(open_set, (tentative_g, (nx, ny)))

return None
```

3.2 Имплементация на A* с евристика

```
def heuristic(a, b):
    """Manhattan distance heuristic for A*."""
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def astar_path(start, goal, terrain_map):
    """
    Find shortest path using A* algorithm.
    Uses Manhattan distance as heuristic.
    """
    import heapq

    open_set = [(heuristic(start, goal), start)]
    open_set_lookup = {start}
    g_score = {start: 0}
    came_from = {}
    visited = set()

    while open_set:
        _, current = heapq.heappop(open_set)
        open_set_lookup.discard(current)

        if current in visited:
            continue
        visited.add(current)

        if current == goal:
            path = []
            node = current
            while node in came_from:
                path.append(node)
                node = came_from[node]
            path.append(start)
            return path[::-1]

        x, y = current
        for dx, dy in [(0, -1), (0, 1), (-1, 0), (1, 0)]:
            nx, ny = x + dx, y + dy

            if not (0 <= nx < MAP_WIDTH and 0 <= ny < MAP_HEIGHT):
                continue
            if terrain_map[ny][nx] != 1:
                continue
```

```
            if (nx, ny) in visited:
                continue

            tentative_g = g_score[current] + 1

            if tentative_g < g_score.get((nx, ny), float('inf')):
                came_from[(nx, ny)] = current
                g_score[(nx, ny)] = tentative_g
                f_score = tentative_g + heuristic((nx, ny), goal)
                if (nx, ny) not in open_set_lookup:
                    heapq.heappush(open_set, (f_score, (nx, ny)))
                    open_set_lookup.add((nx, ny))

    return None
```

3.3 Имплементация на Белман-Форд

```
def bellman_ford_path(start, goal, terrain_map):
    """
    Find shortest path using Bellman-Ford algorithm.
    Relaxes all edges V-1 times.
    """
    # Build list of all edges from walkable tiles
    edges = []
    vertices = set()
    for y in range(MAP_HEIGHT):
        for x in range(MAP_WIDTH):
            if terrain_map[y][x] == 1:
                vertices.add((x, y))
                for dx, dy in [(0, -1), (0, 1), (-1, 0), (1, 0)]:
                    nx, ny = x + dx, y + dy
                    if 0 <= nx < MAP_WIDTH and 0 <= ny < MAP_HEIGHT:
                        if terrain_map[ny][nx] == 1:
                            edges.append(((x, y), (nx, ny), 1))

    # Initialize distances
    dist = {v: float('inf') for v in vertices}
    dist[start] = 0
    came_from = {}

    # Relax all edges V-1 times
    for _ in range(len(vertices) - 1):
        updated = False
        for u, v, w in edges:
            if dist[u] + w < dist[v]:
```

```
                dist[v] = dist[u] + w
                came_from[v] = u
                updated = True
        if not updated:
            break

    # Reconstruct path
    if dist[goal] == float('inf'):
        return None

    path = []
    node = goal
    while node != start:
        path.append(node)
        node = came_from[node]
    path.append(start)
    return path[::-1]
```

4. Демонстрация на работа

Входни данни:

- Карта на терена (14x10 клетки) – дефинирана в constants.py
- Начална позиция на героя – долен ляв ъгъл (1, 8)
- Целева позиция – избрана с клик на мишката

Изходни данни:

- Визуализация на процеса на търсене в реално време
- Намереният най-кратък път (показан в синьо)
- Брой посетени върхове
- Време за изпълнение (в милисекунди)

Демо режимът се активира с бутона "DEMO MODE" или клавиш D. В този режим алгоритъмът се визуализира стъпка по стъпка с настройваема скорост (200ms между стъпките). Потребителят може да избира между трите алгоритъма чрез бутона "Algo".

Контроли на играта:

- Стрелки – движение на героя в 4 посоки
- Клик с мишката – автоматично намиране на път до избраната клетка
- D – превключване на демо режим, E – край на хода

5. Заключение

В проекта успешно са имплементирани три алгоритъма за намиране на най-кратък път: Дийкстра, A* и Белман-Форд. Визуализацията в контекста на играта PetWars позволява да се наблюдава работата на алгоритмите в реално време и да се сравни тяхната ефективност.

Основни резултати:

- A* е най-бърз за насочено търсене към конкретна цел благодарение на евристичната функция
- Дийкстра изследва по-голяма област, но гарантира оптималност във всички случаи
- Белман-Форд е по-бавен поради O(V*E) сложност, но работи с по-общи случаи

Възможности за бъдещи разработки:

- Добавяне на различни тегла на терена (вода, планини, пътища)
- Имплементация на Jump Point Search за още по-бързо търсене в решетки
- Паралелизация на алгоритмите за по-големи карти
- Добавяне на диагонално движение с коефициент √2

6. Използвана литература

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. "Introduction to Algorithms", 3rd Edition, MIT Press, 2009

2. Russell, S., Norvig, P. "Artificial Intelligence: A Modern Approach", 4th Edition, Pearson, 2020

3. Pygame Documentation - https://www.pygame.org/docs/

4. Red Blob Games - Introduction to A* - https://www.redblobgames.com/pathfinding/a-star/

5. Dijkstra, E. W. "A note on two problems in connexion with graphs", Numerische Mathematik, 1959