Aplikasi Algoritma A-star Pathfinding dalam Pembuatan Bot Permainan Snake

Eugene Yap Jin Quan - 13521074
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail : 13521074@std.stei.itb.ac.id

Abstrak—Permainan *Snake* adalah salah satu permainan video yang sangat dikenal, sedangkan algoritma A-star adalah salah satu algoritma pencarian rute. Algoritma A-star menggunakan nilai heuristik untuk menghitung fungsi evaluasi nilai sebuah simpul pencarian. Permainan *Snake* merupakan permainan video sederhana yang merepresentasikan pemain sebagai garis, dengan objektif mendapatkan makanan sebanyak mungkin sebelum menabrak rintangan. Pemetaan komponen permainan Snake pada persoalan pencarian rute adalah kepala *Snake* sebagai simpul asal dan makanan sebagai simpul tujuan. Pada makalah ini, penulis mengimplementasikan permainan *Snake* dan *bot* sederhana menggunakan bahasa Python dan *library* Pygame. Hasil implementasi diujikan menggunakan enam kali. Hasil yang diperoleh adalah rata-rata peroleh poin sebesar 90. Kesimpulan yang penulis peroleh adalah bahwa algoritma A-star dapat digunakan untuk membuat *bot* permainan *Snake*.

Kata kunci—Snake; A-star; rute; bot; Python

I. PENDAHULUAN

Permainan *Snake* adalah salah satu permainan video yang sangat dikenal. Permainan *Snake* berasal dari dunia permainan arkade bernama *Blockade*, sebuah permainan dua pemain yang dirilis pada 1976. Permainan *Snake* kemudian dipopulerkan oleh penjualan ponsel Nokia pada 1997 (Nokia 6110) [1].

Algoritma A* adalah salah satu pilihan algoritma yang dapat digunakan untuk menyelesaikan persoalan pencarian rute dalam graf. Algoritma A* adalah algoritma yang secara optimal dan efisien menelusuri simpul-simpul dalam graf persoalan [2]. Algoritma A* bersifat optimal dan *complete* [3].

Gambar 1. Permainan *Blockade* (1976). Sumber: https://www.arcade-museum.com/game_detail.php?game_id=7160

Ketertarikan penulis terhadap permainan *Snake* dan algoritma pencarian rute mendorong penulis untuk mengeksplorasi penggunaan algoritma A* pada permainan *Snake*. Penulis memutuskan untuk membuat *bot* permainan *Snake* yang dapat menggerakkan pemain secara otomatis menggunakan algoritma A*.

II. LANDASAN TEORI

A. *Algoritma A**

Algoritma A* (A-star) adalah salah satu algoritma pencarian dalam graf. Algoritma A* sering digunakan dalam pencarian rute dan traversal graf. Algoritma ini adalah perluasan dari algoritma Dijkstra [2].

Dalam proses pencarian, algoritma A* memanfaatkan nilai heuristik (estimasi nilai simpul terhadap tujuan) dalam perhitungan fungsi evaluasi. Fungsi evaluasi ini digunakan untuk menentukan urutan pencarian simpul. Secara umum, rumus fungsi evaluasi A* diberikan oleh persamaan berikut [3].

$$f(n) = g(n) + h(n) \quad (1)$$

Sebagai keterangan untuk (1), *f(n)* adalah fungsi evaluasi dan nilai estimasi biaya total dari simpul n ke simpul tujuan. Sementara itu, *g(n)* dan *h(n)* masing-masing secara berturut-turut adalah biaya total dari simpul asal hingga simpul n dan estimasi biaya dari simpul n menuju simpul tujuan [3].

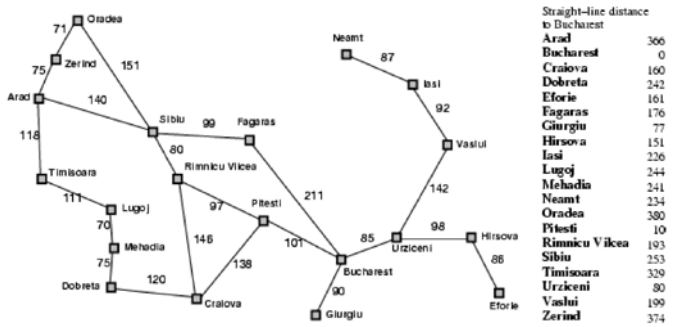
Pemilihan heuristik *h(n)* untuk (1) harus tepat dan tidak bersifat *overestimate*. Apabila heuristik *h* tidak *overestimate*, heuristik tersebut dikatakan *admissible* dan mampu menghasilkan solusi optimal jika digunakan dalam fungsi evaluasi [3].

Secara umum, heuristik yang digunakan dalam algoritma A* adalah *euclidean distance* dan *manhattan distance*. Rumus kedua pilihan untuk ruang 2D diberikan oleh (2) dan (3) secara berturut-turut [2].

$$h = \sqrt{(x_{start} - x_{destination})^2 + (y_{start} - y_{destination})^2} \quad (2)$$

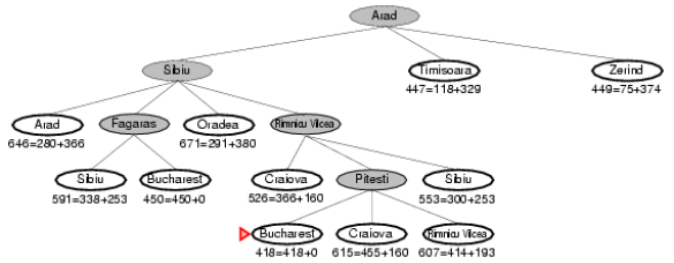
$$h = |x_{start} - x_{destination}| + |y_{start} - y_{destination}| \quad (3)$$

Salah satu penggunaan algoritma A* adalah pencarian rute antarkota. Berikut adalah contoh graf yang merepresentasikan peta kota.



Gambar 2. Representasi Graf dari Peta Romania [4]

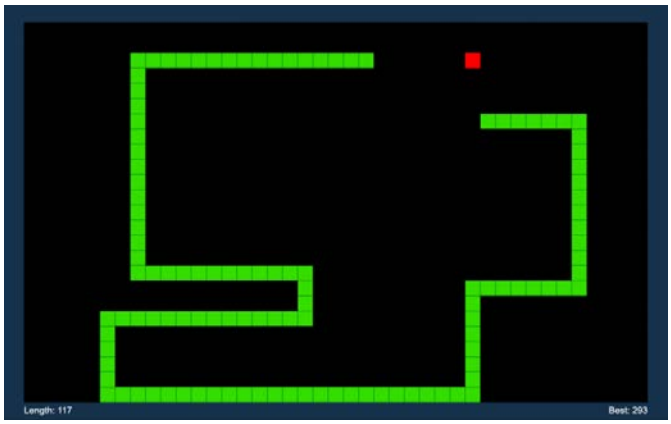
Untuk Gambar 2., solusi pencarian rute dari Arad ke Bucharest dapat digambarkan dengan pohon pencarian berikut.



Gambar 3. Pohon Pencarian dan Solusi Pencarian Rute Arad ke Bucharest [3]

B. *Permainan Snake*

Snake adalah permainan video sederhana yang merepresentasikan pemain sebagai ular atau garis sederhana. Dalam permainan *Snake* sederhana (satu pemain), pemain dapat bergerak pada layar dengan arah kiri, kanan, atas, dan bawah. Objektif dari permainan ini adalah mendapatkan makanan sebanyak mungkin. Apabila sebuah unit makanan diperoleh, karakter ular pemain akan bertambah panjang. Permainan *Snake* berakhir apabila kepala ular menabrak dinding, rintangan, atau bagian tubuh sendiri [5].



Gambar 4. Tampilan Permainan *Snake* Sederhana [1]

III. PEMBAHASAN

A. *Perancangan Spesifikasi Permainan Snake*

Pada eksperimen ini, penulis mengimplementasikan sebuah permainan *Snake* sederhana menggunakan bahasa *python*. Implementasi ini menggunakan varian sederhana dari permainan *Snake*, yaitu beberapa poin berikut.

- Jumlah pemain adalah pemain tunggal.
- Pemain hanya dapat bergerak dalam ruang 2D dengan arah kiri, kanan, atas, atau bawah.
- Pemain hanya dapat berbelok 90 derajat atau bergerak maju.
- Peta berbentuk *grid*, yaitu sususan beberapa titik.
- Tipe makanan hanya satu, sehingga perpanjangan karakter ular akibat konsumsi makanan adalah sama untuk semua makanan.
- Hanya ada satu makanan yang muncul di layar pada saat bersamaan.
- Pembangkitan posisi makanan dilakukan secara acak.
- Permainan menampilkan skor sebagai jumlah makanan yang telah dikonsumsi oleh pemain.

Untuk kebutuhan pengujian, penulis juga menambahkan spesifikasi tambahan. Spesifikasi tambahan ini adalah permainan dapat menerima sebuah *seed* untuk pembangkitan posisi makanan.

B. *Perancangan Bot dan Pemetaan Solusi Persoalan*

Dalam konteks pencarian rute, peta *grid* pada permainan *Snake* dapat digunakan sebagai graf persoalan. Dengan kata lain, graf pada persoalan ini adalah *grid* layar dari permainan, dan titik-titik pada *grid* merupakan simpul-simpul pada graf. Meskipun demikian, beberapa titik dapat berperan sebagai simpul rintangan, yakni dinding dan segmen-segmen tubuh karakter ular.

Sebuah persoalan pencarian rute membutuhkan beberapa informasi, seperti simpul-simpul dan jarak. Pada permainan ini, simpul tujuan adalah setiap titik makanan yang muncul secara bergantian. Adapun simpul asal pencarian, yaitu posisi kepala dari karakter ular. Untuk informasi jarak, jarak antara dua simpul bertetangga dapat ditetapkan sebagai 1 unit. Sementara itu, informasi jarak heuristik dapat menggunakan perhitungan *euclidean distance* (rumus (2)).

Persoalan pencarian rute mungkin tidak menghasilkan solusi apabila tidak terdapat rute. Oleh karena itu, penulis memilih pendekatan untuk menggerakkan karakter ular menuju simpul aman terdekat. Sesudah bergerak, *bot* akan mengulangi proses pencarian rute.

Berdasarkan uraian di atas, penggunaan algoritma A* dalam proses *bot* adalah mencari rute dari titik kepala ular saat itu menuju sebuah titik makanan. Proses pencarian ini akan dipanggil setiap kali sebuah makanan diletakkan pada peta. *Bot* ini juga akan mengatasi kasus solusi nil.

C. *Implementasi Rancangan*

Implementasi rancangan menggunakan Python dan *library* Pygame. Implementasi permainan *Snake* dilakukan dengan melakukan pengembangan terhadap tutorial pembuatan *Snake* sederhana [6]. Sementara itu, implementasi komponen bot adalah sebagai berikut. Implementasi lengkap dapat dilihat pada tautan https://github.com/yuujin-Q/stima-snake-bot.

```python
import math


class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __ne__(self, other):
        return not (self == other)

    def __gt__(self, other):
        return self.x > other.x and self.y > other.y

    def __ge__(self, other):
        return self.x ≥ other.x and self.y ≥ other.y

    def __lt__(self, other):
        return self.x < other.x and self.y < other.y

    def __le__(self, other):
        return self.x ⩽ other.x and self.y ⩽ other.y

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Point(self.x - other.x, self.y - other.y)

    def __hash__(self):
        return hash((self.x, self.y))

    def to_string(self):
        return f"({self.x}, {self.y})"

    def get_x(self):
        return self.x

    def get_y(self):
        return self.y

    @staticmethod
    def manhattan_distance(p1, p2):
        difference = p1 - p2
        return abs(difference.get_x()) + abs(difference.get_y())

    @staticmethod
    def euclidean_distance(p1, p2):
        return math.sqrt((p1.get_x() - p2.get_x()) ** 2 + (p1.get_y() - p2.get_y()) ** 2)

```

Gambar 5. Implementasi Point (point.py)

```python
class SearchNode:
    def __init__(self, search_point, path_list, path_cost):
        self.search_point = search_point
        self.path_list = [direction for direction in path_list]
        self.path_cost = path_cost

    def add_self_to_movement_list(self):
        self.path_list = self.path_list + [self.search_point]

    def __lt__(self, other):
        return self.path_cost < other.path_cost

    def get_search_point(self):
        return self.search_point

    def get_path_list(self):
        return self.path_list[:]

    def get_path_cost(self):
        return self.path_cost

```

Gambar 6. Implementasi SearchNode (searchnode.py)

```python
from logic.point.point import Point


class GridMap:
    def __init__(self, width, height, obstacle_list, pixel_size=10):
        self.adjacency_list = {}

        for i in range(0, width, pixel_size):
            for j in range(0, height, pixel_size):
                grid_tile = Point(i, j)
                neighbors = []

                # Add Left, Right, Up, Down Neighbors
                adjacent = [grid_tile + Point(-pixel_size, 0),
                            grid_tile + Point(pixel_size, 0),
                            grid_tile + Point(0, -pixel_size),
                            grid_tile + Point(0, pixel_size)]

                # Borders
                top_left = Point(0, 0)
                bottom_right = Point(width, height)

                for points in adjacent:
                    if top_left ⩽ points < bottom_right and points not in obstacle_list:
                        neighbors.append(points)

                self.adjacency_list[grid_tile] = neighbors

    def get_neighbors(self, point):
        if point in self.adjacency_list:
            return self.adjacency_list[point]
        else:
            return []

```

Gambar 7. Implementasi GridMap (gridmap.py)

```python
import pygame
from queue import PriorityQueue
from logic.snakebot.models.searchnode import SearchNode
from logic.snakebot.models.gridmap import GridMap
from logic.point.point import Point


class SnakeBot:
    solution_cost = float("inf")
    solution_path = []
    show_debug = False

    def __init__(self, screen: pygame.display, snake_body, pixel_size, show_debug=False):
        self.screen = screen
        screen_width, screen_height = screen.get_size()
        self.graph = GridMap(screen_width, screen_height, snake_body[:len(snake_body)-1], pixel_size)
        self.show_debug = show_debug
        self.start_point = Point(0, 0)
        self.finish_point = Point(0, 0)

    def plan_route(self, start_point, finish_point):
        # SETUP: priority queue, starting node, explored nodes
        search_pqueue = PriorityQueue()
        starting_node = SearchNode(start_point, [], 0)
        self.start_point = start_point
        self.finish_point = finish_point

        # enqueue start_node:
        node_priority = starting_node.get_path_cost() + Point.euclidean_distance(start_point, finish_point)

        search_pqueue.put((node_priority, starting_node))
        explored_points = set()

        # SEARCH: do search loop
        while not search_pqueue.empty():
            # dequeue current node to search
            node_priority, current_node = search_pqueue.get()
            current_node.add_self_to_movement_list()

            # show debug messages
            if self.show_debug is True:
                file_path = "log.txt"  # Replace with your desired file path
                file = open(file_path, "a+")

                file.write(str(node_priority) + "---" + current_node.get_search_point().to_string()
                           + str(current_node.get_path_cost()))
                for move in current_node.get_path_list():
                    file.write("   ")
                    file.write(move.to_string() + ",\n")
                file.write("\n")

                # Close the file
                file.close()

            # Goal check: return from function if goal is met
            if current_node.get_search_point() == finish_point:
                self.solution_path = current_node.get_path_list()
                self.solution_cost = current_node.get_path_cost()

                print("Solution found")
                return

            if current_node.get_search_point() in explored_points:
                continue

            # enqueue neighbors
            for neighbor_point in self.graph.get_neighbors(current_node.get_search_point()):
                if neighbor_point not in explored_points:
                    neighbor_cost = current_node.get_path_cost() + \
                                    Point.manhattan_distance(current_node.get_search_point(), neighbor_point)
                    # distance between neighbors in grid is same as manhattan distance

                    new_neighbor_node = SearchNode(neighbor_point, current_node.get_path_list(), neighbor_cost)

                    # enqueue neighbor to search queue
                    node_priority = neighbor_cost + Point.euclidean_distance(neighbor_point, finish_point)

                    search_pqueue.put((node_priority, new_neighbor_node))

            # mark current node as visited
            explored_points.add(current_node.get_search_point())

        # Search failed
        print("Search failed")

    def get_movement_list(self):
        if len(self.solution_path) < 1:
            safe_neighbors = self.graph.get_neighbors(self.start_point)
            if len(safe_neighbors) > 0:
                return [safe_neighbors[0] - self.start_point]
            else:
                return []
        else:
            return [self.solution_path[i] - self.solution_path[i-1] for i in range(1, len(self.solution_path))]

```

Gambar 8. Implementasi Algoritma Pencarian A* (snakebot.py)

Terkait implementasi permainan dan *bot*, berikut adalah beberapa keterangan tambahan.

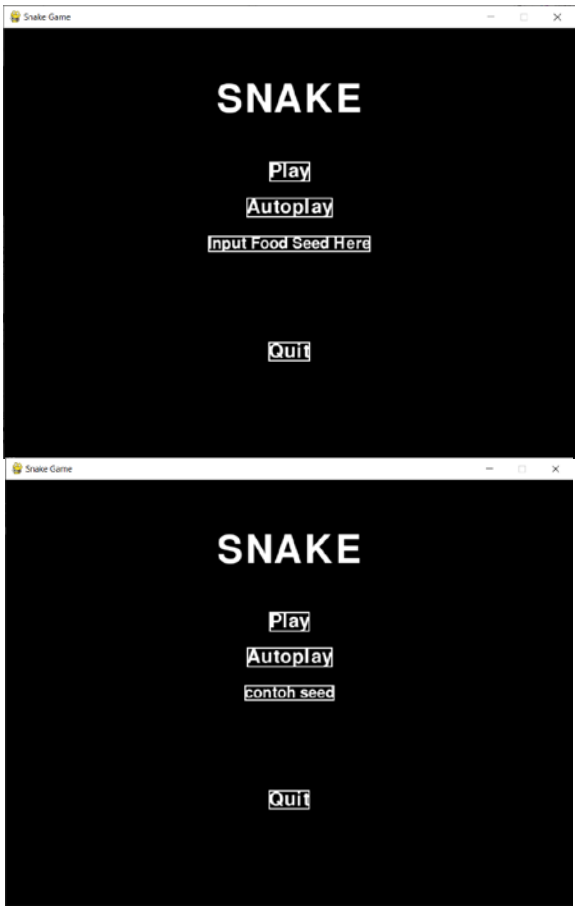
- Apabila pencarian tidak menghasilkan sebuah rute, gerakan yang diberikan oleh bot kepada permainan adalah arah menuju simpul tetangga terdekat. Simpul tetangga yang dimaksud adalah simpul aman yang tidak memiliki rintangan. Simpul tetangga ini dibangkitkan dengan urutan kiri, kanan, atas, bawah.
- Solusi dari pencarian merupakan sebuah *list* urutan titik-titik yang dikunjungi. Solusi ini kemudian diterjemahkan menjadi *list* urutan vektor arah pergerakan antartitik untuk digunakan oleh program utama.
- Ukuran layar permainan adalah 800x600 pixel, dengan satu unit ukuran objek permainan adalah 25x25 pixel. Dengan demikian, *grid* pada permainan secara efektif berukuran 32x24 objek.
- Karakter ular pada permainan ini merupakan *list* dari titik-titik posisi segmen tubuh (termasuk kepala). Setiap segmen diwarnai dengan warna hijau. Apabila permainan berakhir, posisi terakhir kepala ular akan berubah menjadi warna putih.
- Setiap objek makanan diwarnai dengan warna merah.
- Poin dihitung berdasarkan jumlah makanan yang berhasil diambil.

D. Pengujian Hasil Implementasi

Hasil implementasi ini diuji menggunakan beberapa *seed* pengujian. *Seed* ini dapat dimasukkan pada halaman utama permainan. Kriteria pengujian yang dilakukan adalah mengamati pergerakan dan jumlah poin yang berhasil diperoleh oleh *bot*.

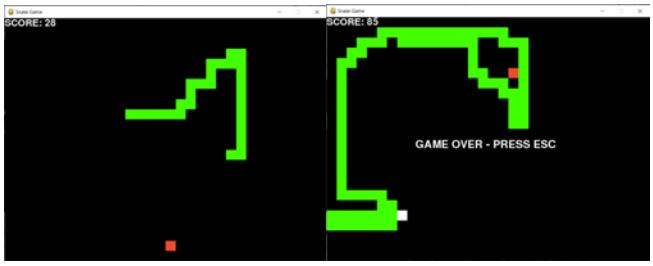
Langkah pengujian dari hasil implementasi adalah sebagai berikut.

1. Membuka aplikasi permainan *Snake.*
2. Menekan kotak “Input Food Seed Here”, kemudian mengetik *seed* yang diinginkan.
3. Menekan tombol “Autoplay” untuk memulai pengujian.
4. Mengamati pengujian dengan menekan tombol spasi yang menghentikan dan melanjutkan pengujian.

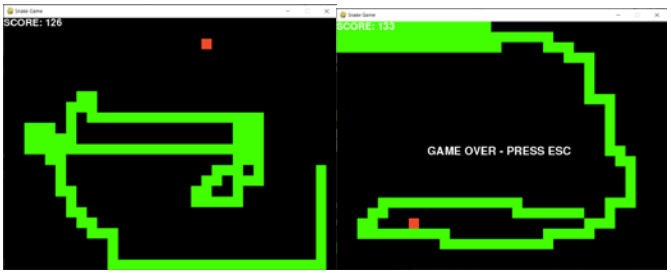


Gambar 9. Tampilan Utama beserta Contoh Masukan *Seed*

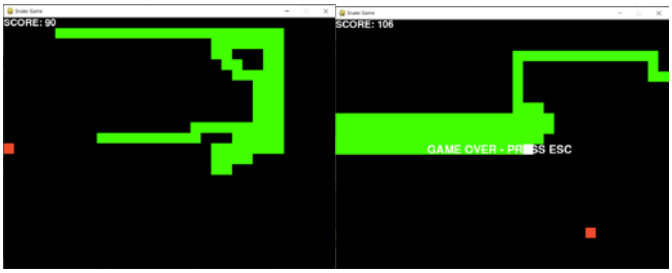
Dengan melakukan langkah pengujian tersebut dan dengan menggunakan *seed* “stima”, “ular”, “xenzia”, “game”, “snake”, dan “strategi”, hasil pengujian yang diperoleh adalah sebagai berikut.



Gambar 10. *Seed* “stima”. Poin akhir 85.

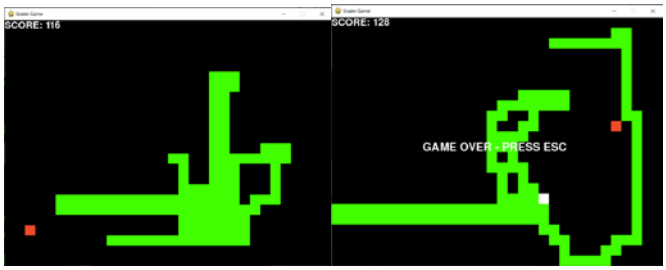


Gambar 11. *Seed* “ular”. Poin akhir 133.



Gambar 12. *Seed* “xenzia”. Poin akhir 106.

Gambar 13. *Seed* “game”. Poin akhir 1.



Gambar 14. *Seed* “snake”. Poin akhir 128.

Gambar 15. *Seed* “strategi”. Poin akhir 92.

E. Analisis Hasil Pengujian

Berdasarkan hasil pengujian, implementasi *bot* permainan *Snake* telah berhasil menggerakkan pemain untuk beberapa saat. Rata-rata perolehan poin untuk keenam pengujian adalah 90 poin. Nilai maksimal dari perolehan poin untuk pengujian ini adalah 133 poin. Di sisi lain, nilai minimal adalah 1 poin. Nilai minimal ini merupakan pengecualian. Pada kasus uji *seed* “game”, *Snake* keluar dari layar pada arah kanan setelah memakan satu unit makanan. Hal ini adalah perilaku abnormal yang mungkin disebabkan oleh kesalahan implementasi. Hal ini diamati terjadi pada *seed-seed* lain juga.

Dari lima *seed* berbeda (selain *seed* “game”), permainan berhenti akibat kepala *Snake* mencapai ujung layar ataupun menabrak segmen tubuh *Snake*. Hal ini disebabkan oleh kondisi layar yang menyebabkan tidak ditemukannya solusi pencarian rute. Pada implementasi, penanganan solusi nil dilakukan dengan bergerak menuju simpul tetangga terdekat dengan urutan tertentu. Pilihan implementasi ini pada beberapa kasus mampu mengulur waktu hingga tercapainya sebuah solusi. Akan tetapi, akibat urutan tetap pemilihan tetangga, *Snake* mungkin saja memilih tetangga yang berakibat pada rute jalan buntu yang lebih singkat.

Sebagai perbandingan, jumlah panjang maksimal untuk konfigurasi implementasi permainan *Snake* ini adalah 768 poin (32x24, diperoleh dari ukuran layar). Akan tetapi, nilai maksimal ini susah untuk dicapai secara nyata. Hal ini dikarenakan pergerakan *bot Snake* hanya mengutamakan pencarian rute menuju makanan saat itu. Dengan demikian, pergerakan *bot* pada proses pencarian rute tersebut mungkin menghalangi proses pencarian berikutnya. Hal ini sangat mungkin terjadi apabila *Snake* sudah mencapai panjang yang signifikan.

IV. Simpulan

Berdasarkan uraian di atas, algoritma A* terbukti dapat digunakan untuk membuat sebuah *bot* permainan *Snake*. Menurut sampel pengujian, *bot* hasil implementasi mampu mencapai rata-rata 90 poin sebelum permainan berakhir. Berdasarkan hasil pengujian juga, penulis menyimpulkan bahwa implementasi ini masih bisa dikembangkan lebih lanjut. Salah satu aspek yang bisa dikembangkan lagi adalah penanganan solusi nil pada proses pencarian rute. Penulis berpendapat bahwa penanganan solusi nil yang lebih baik mampu meningkatkan lama waktu permainan *Snake* berlangsung.

Link Video Youtube

Video penjelasan singkat mengenai topik dan eksperimen yang dilakukan pada pembuatan makalah ini tersedia pada link https://youtu.be/-MLEq3P-vgA.

Ucapan Terima Kasih

Puji syukur penulis panjatkan kepada Tuhan Yang Maha Esa karena dengan berkat-Nya, penulis dapat menyelesaikan

makalah ini dengan baik dan tepat waktu. Penulis ingin mengucapkan terima kasih kepada Dr. Nur Ulfa Maulidevi, S.T., M.Sc. selaku dosen Kelas 2 Mata Kuliah IF2211 Strategi Algoritma 2022/2023 karena telah membimbing dan mengajari penulis tentang algoritma pencarian rute. Penulis juga ingin mengucapkan terima kasih kepada pencipta referensi-referensi yang penulis gunakan dalam penyelesaian makalah ini. Penulis juga mengucapkan terima kasih kepada kedua orang tua penulis yang telah mendukung penulis dalam proses penyusunan makalah.

Penulis bersyukur karena mendapatkan kesempatan untuk melakukan eksplorasi dan eksperimen mandiri terhadap topik ini. Penulis berharap bahwa makalah ini dapat bermanfaat bagi pembaca.

REFERENSI

[1] G. Bateson. “The History of Snake – From the Arcade to Now,” 2023. Accessed: May 20, 2023. [Online]. Available: https://www.coolmathgames.com/blog/the-history-of-snake-the-game

[2] Brilliant.org, “A* Search,” 2016. Accessed: May 20, 2023. [Online]. Available: https://brilliant.org/wiki/a-star-search/.

[3] N. U. Maulidevi, dan R. Munir, “Penentuan Rute (Route/Path Planning),” 2021. Accessed: May 20, 2023. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf

[4] N. U. Maulidevi, dan R. Munir, “Penentuan Rute (Route/Path Planning)” 2021. Accessed: May 20, 2023. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf

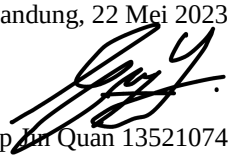
[5] D. Kosbie, “Snake Tutorial,” 2010. Accessed: May 20, 2023. Available: https://www.kosbie.net/cmu/fall-10/15-110/handouts/snake/snake.html

[6] W. Urooj, “Snake Game in Python | Snake Game Program using Pygame,” 2023. Accessed: May 21, 2023. [Online]. Available: https://www.edureka.co/blog/snake-game-with-pygame/

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 22 Mei 2023



Eugene Yap Jin Quan 13521074