TELEKONTRAN, VOL. 9, NO. 2, OKTOBER 2021
DOI : 10.34010/telekontran.v9i2.5635
p-ISSN : 2303 – 2901
e-ISSN : 2654 – 7384

Analisis Perbandingan Algoritma Breadth First Search (BFS) dan Algoritma A*

Comparative Analysis of Breadth First Search (BFS) Algorithm and A Algorithm*

Valdyna Aditiya*, Budi Herdiana
Program Studi Teknik Elektro, Fakultas Teknik dan Ilmu Komputer
Universitas Komputer Indonesia Jl. Dipati ukur No 112, Bandung
*Email : valdyna@mahasiswa.unikom.ac.id

Abstrak – Pada *path planning* terdapat beberapa metode yang dapat digunakan. Metode yang dapat digunakan untuk *path planning* yaitu algoritma *Breadth First Search* (BFS) dan algoritma A*. Metode yang digunakan tergantung dari keadaan lingkungan. Pengujian *path planning* pada berbagai lingkungan dengan menggunakan metode yang berbeda dapat mengetahui performansi tiap metode tersebut. Sehingga tujuan penelitian ini adalah untuk membandingkan antara algoritma BFS dengan algoritma A* dalam melakukan *path planning*. Proses perbandingan dilakukan pada lima lingkungan pengujian. Lingkungan pengujian yang digunakan yaitu tanpa *obstacle*, *obstacle trap*, *obstacle* sederhana, *obstacle maze* dan *obstacle narrow*. Perbandingan algoritma BFS dan algoritma A* berdasarkan waktu eksekusi dan jumlah *node* yang dibutuhkan untuk melakukan *path planning*. Hasil penelitian menunjukkan bahwa pencarian jalur dari titik *start* ke titik *goal* dapat diselesaikan dengan algoritma BFS dan A*. Pada algoritma BFS maupun A* menghasilkan biaya jalur yang sama. Perbedaan terjadi pada *node-node* yang dibutuhkan algoritma BFS dan A* untuk menghasilkan jalur dari titik *start* hingga titik *goal*. Algoritma BFS membutuhkan *node* lebih banyak dibandingkan dengan A* untuk mencapai titik *goal*. Perbedaan jumlah *node* tersebut membuat waktu eksekusi menjadi berbeda. Waktu eksekusi pada algoritma BFS membutuhkan waktu lebih banyak dibandingkan dengan A*. Berdasarkan pengujian yang telah dilakukan maka algoritma A* lebih cepat dalam melakukan *path planning.* Tetapi pada lingkungan pengujian *maze* terjadi perbedaan waktu yang sedikit. Pada BFS memerlukan waktu tercepat 4,09 detik serta pada A* memerlukan waktu tercepat 3,88 detik. Serta pada lingkungan *maze* memiliki perbedaan jumlah *node* cukup sedikit yaitu 26 *node*. Hal tersebut membuktikan bahwa A* tidak selalu unggul jauh dengan BFS.

Kata kunci : A*, *Breadth First Search* (BFS), lingkungan pengujian, *node*, waktu.

Abstract *- In path planning there are several methods that can be used. The methods that can be used for path planning are the Breadth First Search (BFS) algorithm and the A* algorithm. The method used depends on the environmental conditions. Path planning testing in various environments using different methods can determine the performance of each method.. So the purpose of this study is to compare the BFS algorithm with the A* algorithm in doing path planning. The comparison process was carried out in five test environments. The test environment used is without obstacles, obstacle traps, simple obstacles, maze obstacles and narrow obstacles. Comparison of the BFS algorithm and the A* algorithm based on the execution time and the number of nodes required to perform path planning. The results show that the path search from the starting point to the destination point can be completed using the BFS and A* algorithms. Both the BFS and A* algorithms produce the same path costs. The difference occurs in the nodes required by the BFS and A* algorithms to generate a path from the starting point to the destination point. The BFS algorithm requires more nodes than A* to reach the destination point. The difference in the number of nodes makes the execution time different. The execution time of the BFS algorithm takes longer than A*. Based on the tests that have been carried out, the A* algorithm is faster in planning paths. But in the maze test environment there is a slight time difference. On BFS it takes the fastest time of 4.09 seconds and on A* it takes the fastest time of 3.88 seconds. And in the maze environment, there are quite a few differences in the number of nodes, namely 26 nodes. This proves that A* is not always far superior to BFS.*

Keywords *: A*, Breadth First Search (BFS), node, test environment, time.*

I. PENDAHULUAN

A. Latar Belakang

Robot yang dapat bergerak secara mandiri membutuhkan sistem navigasi yang dapat membantu robot bergerak hingga ke tujuan. Salah satu sistem untuk membantu navigasi robot adalah *path planning*. *Path planning* merupakan suatu pencarian jalur yang nantinya akan dilalui oleh robot. Pada proses pencarian jalur haruslah memilih jalur yang terdekat [1]. Pada proses *path planning* robot akan menghindari semua rintangan sepanjang jalan [2]. Pada *path planning* dapat diaplikasikan pada robot otonom. Robot otonom memiliki memampuan untuk bekerja secara otomatis tanpa campur tangan manusia. Robot otonom dapat bekerja untuk waktu yang lama, serta dapat bekerja pada tempat yang berbahaya.

Path planning juga dapat diaplikasikan pada perancangan rute transportasi umum, sehingga mencarikan rute tercepat untuk sampai tujuan [1]. *Path planning* juga dapat diaplikasikan pada robot pengangkut barang yang akan membawa barang dari titik asal hingga titik akhir dengan rute tercepat [2]. Serta dapat digunakan pada *unmanned aerial vehicle*, dengan melakukan pemetaan mendetail pada bagian daratan saja dan mengecualikan bagian perairan [3]. Pada dunia industri *path planning* juga dapat diaplikasikan, seperti pada *hand* robot untuk memindahkan barang dari satu titik ke titik lainnya [4]. *Path planning* dapat melakukan pemburuan atau pengejaran terhadap robot lain dengan terdapat beberapa rintangan [5].

Tentu dalam melakukan *path planning* harus memiliki keamanan sehingga tidak menabrak kendaraan lain [6]. Pada proses pencarian jalur harus dapat menghindari setiap *obstacle* yang ada [7][8]. Menghindari *obstacle* akan membuat hasil pembuatan jalur tidak menabrak dengan *obstacle*. Pada proses *path planning* juga harus dapat dilakukan dengan waktu yang secepat mungkin [9]. Pada algoritma *path planning* haruslah mampu mengoptimalkan kualitas jalur dan cukup cepat untuk digunakan dalam aplikasi kompleks dunia nyata [10]. Kualitas jalur yang baik akan membuat robot dapat bekerja dengan optimal.

Setiap metode pencarian jalur bergantung pada keadaan lingkungan [11]. Keadaan lingkungan mewakili semua posisi yang mungkin untuk orientasi robot. Pada perencanaan jalur terdapat beberapa algoritma yang dapat digunakan. *Breadth First Search* (BFS) merupakan salah satu algoritma yang dapat melakukan *path planning*. Serta algoritma A* merupakan suatu program yang dapat digunakan dalam *path planning* [9].

B. Tinjauan *State of Art*

Algoritma A* merupakan suatu algoritma perbaikan dari BFS, dengan modifikasi dari fungsi *heuristic*. Pada algoritma A* mempunyai prinsip yang sama dengan BFS, hanya berbeda pada proses pencarian titik tujuan dengan mempertimbangkan jaraknya. Pada proses *path planning* BFS akan melakukan pencarian menyeluruh atau mengecek dari *node* awal ke semua *node* tetangganya secara terurut [12]. Sedangkan pada A* akan melakukan pencarian dengan memilih *node* baru berdasarkan biaya ke tujuan [5]. Proses *path planning* BFS dan *Rapidly-exploring Random Trees* (RRT) menghasilkan panjang jalur yang berbeda pada tiap lingkungan, pada empat *scenario* BFS menghasilkan jarak lebih pendek dan pada dua *scenario* RRT yang lebih pendek [13]. Pada percobaan BFS dan RRT tidak membandingkan jumlah *node* yang dibutuhkan [13]. Pada pengujian algoritma A* menghasilkan waktu yang lebih cepat dibandingkan dengan algoritma *basic theta** dan *phi** dalam *path planning*, tetapi algoritma A* menghasilkan jarak jalur yang paling jauh [9]. Sehingga jalur yang dihasilkan oleh A* tidak efektif karena menghasilkan jarak yang paling jauh.

Keadaan lingkungan dapat mempengaruhi dalam *path planning*. Sehingga perlunya dilakukan pengujian *path planning* dengan berbagai lingkungan. Dengan melakukan *path planning* pada tiap lingkungan dengan metode berbeda sehingga dapat mengetahui bagaimana peformansinya. Pada pengujian sebelumnya belum ada pengujian komparasi waktu eksekusi algoritma BFS dan A* pada berbagai lingkungan. Serta pada pengujian sebelumnya belum ada komparasi jumlah *node* yang dibutuhkan algoritma BFS dan A* sehingga menghasilkan *path planning*.

C. Tujuan

Tujuan penelitian ini adalah untuk membandingkan antara algoritma BFS dan A*. Membandingkan waktu eksekusi yang dibutuhkan untuk melakukan *path planning*. Serta dilakukan perbandingan jumlah *node* yang dibutuhkan sehingga menghasilkan rancangan jalur dari titik *start* hingga titik *goal*. Pengujian dilakukan pada lima lingkungan yaitu lingkungan tanpa *obstacle*, *obstacle trap*, *obstacle* sederhana, *obstacle maze* dan *obstacle narrow*. Pengujian akan dilakukan sebanyak 30 kali pada tiap lingkungan.

D. Sistematika Pembahasan

Pada bagian kedua yaitu metodologi akan menjelaskan algoritma yang digunakan serta

library yang digunakan pada python. Algoritma yang digunakan dalam *path planning* yaitu BFS dan A*. Pada bagian 3 yaitu hasil dan pembahasan akan menjelaskan waktu eksekusi tiap algoritma dan jumlah *node* yang dibutuhkan sehingga menghasilkan rancangan jalur. Hasil dari pengujian pada lima lingkungan yang telah dilakukan akan disimpulkan dan disajikan pada bagian 4.

II. METODOLOGI

Rancangan sistem dibuat menggunakan aplikasi python 3.9.8. Program dijalankan menggunakan komputer dengan spesifikasi prosesor intel CORE i5 dengan RAM 16 GB.

A. *Pseudo-code*

Pada program awal akan melakukan inisialisasi titik *start* dan titik *goal*. *Pseudo-code* Algoritma A* dapat dilihat pada **Gambar 1**.

Pseudo-code A* Algorithm

```
function  A_Star (start, goal)
openset[grid(start)]=start
while openset is not empty
      c ← min((openset.cost)+heuristic(goal,openset))
      current ← openset[c]
      if current=goal // end
      remove current from openset
      closedset ← current
      for i to 8
      node=Node(current.x + movement[i][0],
                current.y + movement[i][1],
                current.cost + movement[i][2],c)
      n=grid(node)
      if node not safe, do nothing
      if n in closed, do nothing
      if n not in openset:
         openset[n]=node
      else:
         if openset[n].cost > node.cost
         openset[n]=node
final path
end function
```

Gambar 1. *Pseudo-code* Algoritma A*

Setelah titik *start* ditentukan maka akan melakukan pencarian pada titik *goal*. Proses pencarian dengan menambahkan *node-node* mulai dari titik *start*. Pada proses penambahan *node* akan menghindari area yang tidak aman seperti terdapat *obstacle* pada titik tersebut. Setelah *node-node* mencapai titik *goal* maka akan dilakukan kalkulasi jalur, sehingga menghasilkan jalur terpendek antara titik *start* dan titik *goal*.

B. *Library* Python

Program menggunakan beberapa *library* yaitu *time, math* dan *matplotlib.pyplot as plt.* Pada awal program melakukan *import* terhadap *library* yang akan digunakan, dapat dilihat pada **Gambar 2**.

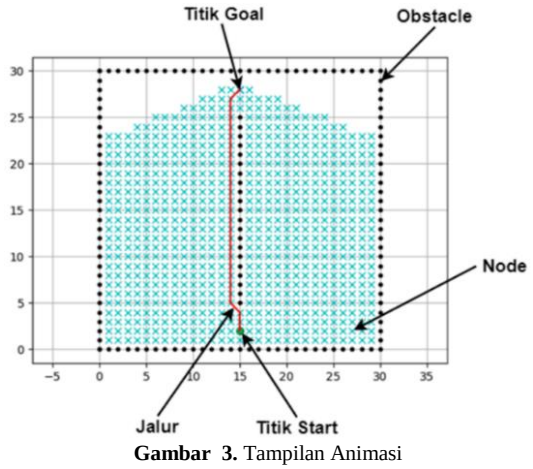
```
import time
import math
import matplotlib.pyplot as plt
show_animation = True
```

Gambar 2. *Library* Python

Library *time* digunakan untuk menghitung waktu yang dibutuhkan dalam melakukan proses *path planning*. Waktu dihitung saat program mulai dijalankan hingga menghasilkan rancangan jalur, dari titik *start* sampai titik *goal*. Waktu yang dihasilkan dalam satuan detik. *Library math* merupakan fungsi yang digunakan untuk melakukan operasi perhitungan aritmatika dan trigonometri. Fungsi perhitungan yang digunakan seperti *math.hypot* dan *math.sqrt*. Fungsi dari *math.sqrt* merupakan perhitungan akar pangkat dua, seperti *math.sqrt(2)* menghasilkan nilai 1.41. Fungsi dari *math.hypot* merupakan metoda norma *Euclidean*. Norma *Euclidean* merupakan jarak dari titik asal ke koordinat yang diberikan. Fungsi dari *math.hypot* digunakan untuk menemukan jarak sisi miring dari segitiga siku – siku yang dapat dihitung dengan menggunakan teorema *Phythagoras*. Penentuan sisi miring dari segitiga siku - siku menggunakan notasi matematika yang dituliskan sebagai:

$$\sqrt{(x * x + y * y)} \quad (1)$$

Pada hasil akhir dari pengujian terdapat visual gambar. Sehingga pada program menggunakan *library matplotlib.pyplot as plt*. *Library* ini digunakan untuk menampilkan seperti titik *start, titik goal, obstacle, node* dan jalur. Contoh dari tampilan animasi dapat dilihat pada **Gambar 3**.



Gambar 3. Tampilan Animasi

Seperti yang dapat dilihat pada **Gambar 2** titik *start* ditandai dengan lingkaran berwarna hijau. Titik *goal* ditandai dengan silang berwarna biru. *Node* ditandai dengan silang berwarna *cyan*. *Obstacle* ditandai titik warna hitam. Serta hasil jalur ditandai dengan garis berwarna merah.

C. Program Utama

Program akan mengidentifikasi posisi *start* dan *goal* sesuai dengan *input* yang telah diberikan. Pencarian jalur akan menambahkan *node* mulai dari titik *start* hingga titik *goal*. Proses pencarian jalur akan menghindari semua *obstacle*. Pada algoritma A* menambahkan *node* baru dengan melihat jarak *node* baru tersebut ke titik *goal*. Program *start* dan *goal position* dapat dilihat pada **Gambar 4**.

```
def planning (self,start_x,start_y,goal_x,goal_y):
    start_position=self.Node(self.xy(start_x,
                   self.min_ox),self.xy(start_y,
                   self.min_oy),0.0,-1)
    goal_position=self.Node(self.xy(goal_x,
                   self.min_ox),self.xy(goal_y,
                   self.min_oy),0.0,-1)
    open_set, closed_set=dict(), dict()
    open_set[self.grid(start_position)]=start_position
```

Gambar 4. *Start* & *Goal Position*

Pada **Gambar 4** program melakukan inisialisasi titik *start* dan titik *goal*, dengan memberikan nilai pada *start_position* dan *goal_position*. Berdasarkan program untuk mendapatkan nilai *start_position* dan *goal_position* menggunakan sub program yaitu *self.xy*. Pada awal program *open_set* diberikan nilai dari *start_position* karena akan menjadi awal pencarian titik *goal*, serta *closed_set* belum diberikan nilai. Setelah melakukan inisialisasi *start* dan *goal* maka proses pencarian dilakukan dengan mencari nilai c_id yang menggunakan fungsi *heuristic*, program dapat dilihat pada **Gambar 5**.

```
c_id = min(
    open_set,
    key=lambda o: open_set[o].cost +
    self.heuristic(goal_position,open_set[o]))
current = open_set[c_id]
```

Gambar 5. Program Nilai c_id

Pada program **Gambar 5** dilakukan proses perhitungan jarak dari suatu *node* ke titik *goal*. Proses tersebut menggunakan rumus:

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

Perhitungan jarak menggunakan fungsi *heuristic*. *Heuristic* akan diberikan nilai dari *goal_position* dan nilai dari *open_set*. Hasil c_id yang didapat maka akan dimasukan sebagai nilai *current*. Setelah *current* diisi oleh nilai yang didapat oleh c_id maka nilai *current* akan diberikan pada *closed_set*. Pada proses pencarian titik *goal* maka *node* akan bertambah pada setiap arah yaitu arah x (kanan atau kiri), arah y (atas atau bawah) dan arah diagonal. Terdapat variabel n_id mendapatkan masuk dari *node* yang sebelumnya *node* tersebut telah dilakukan kalkulasi dengan *self.grid* terlebih dahulu. Bila pada *node* baru tersebut tidak aman maka jangan lakukan aksi apapun kemudian lanjutkan proses. Program pencari arah *node* dapat dilihat pada **Gambar 6**.

```
del open_set[c_id]
closed_set[c_id] = current
for i, _ in enumerate(self.movement):
    node = self.Node(current.x + self.movement[i][0],
                     current.y + self.movement[i][1],
                     current.cost + self.movement[i][2],
                     c_id)
    n_id = self.grid(node)
    if not self.verify(node):
        continue
    if n_id in closed_set:
        continue
    if n_id not in open_set:
        open_set[n_id] = node
    else:
        if open_set[n_id].cost > node.cost:
            open_set[n_id] = node
```

Gambar 6. Program Pencari Arah *Node*

Serta juga bila n_id telah ada pada *closed_set* maka jangan lakukan aksi apapun kemudian lanjutkan proses. Apabila nilai n_id tidak berada pada *open_set* maka masukan nilai tersebut pada *node*. Pada proses pencarian titik *goal node* akan menghindari *obstacle*, sehingga koordinat yang terdapat *obstacle* akan diabaikan. Bila pencarian *node* telah menemukan titik *goal* maka proses pencarian akan berhenti. Penentuan ditemukannya titik *goal* dengan melihat nilai *current* apakah sama dengan nilai *goal_position*. Program penentuan ditemukan titik *goal* dapat dilihat pada **Gambar 7**.

```
if current.x==goal_position.x and current.y==goal_position.y:
    print("Jalur Ditemukan")
    goal_position.main = current.main
    goal_position.cost = current.cost
    print("Panjang Jalur :", current.cost)
    break
```

Gambar 7. Program Penentuan Titik *Goal*

Terdapat nilai *current* x dan y yang dimana merupakan nilai yang didapat dari nilai c_id. Bila nilai *current x* dan *current y* tersebut sama dengan nilai dari *goal position x* dan *goal position y*, maka titik *goal* telah ditemukan dan bila tidak sama maka pencarian akan terus berlanjut. Setelah titik *goal* ditemukan maka proses pencarian akan berhenti. Setelah titik *goal* ditemukan selanjutnya yaitu proses pencarian jalur yang paling dekat. Proses pencarian jalur dengan menggunakan sub program *final_path*. Pada hasil *path planning* akan menghasilkan jalur yang paling optimal dengan jarak paling dekat sesuai dengan *node* – *node* yang telah dibuat dari titik *start* sampai titik *goal*.

Pada program utama terdapat program untuk tampilan animasi. Tampilan animasi akan menampilkan proses *node* dalam melakukan

pencarian titik *goal*. Program *show animation* dapat dilihat pada **Gambar 8**.

```
if show_animation:
    plt.plot(self.position(current.x, self.min_ox),
             self.position(current.y, self.min_oy), "xc")
    plt.gcf().canvas.mpl_connect('key_release_event',
                  lambda event: [exit(
                  0) if event.key == 'escape' else None])
    if len(closed_set.keys()) % 10 == 0:
        plt.pause(0.001)
```

Gambar 8. Program *Show Animation*

Program *show animation* merupakan fungsi yang akan menampilkan posisi *node-node* berdasarkan nilai posisi *grid*. Pada saat program mencari titik *goal* maka akan menampilkan *node-node* yang bertambah hingga mencapai titik *goal*. Untuk nilai x dilakukan kalkulasi dengan menggunakan *self.position* yaitu dengan diberikan nilai dari *current.x* dan *self.min_ox*. Kemudian untuk nilai y menggunakan *self.position* yaitu dengan diberikan nilai dari *current.y* dan *self.min_oy*.

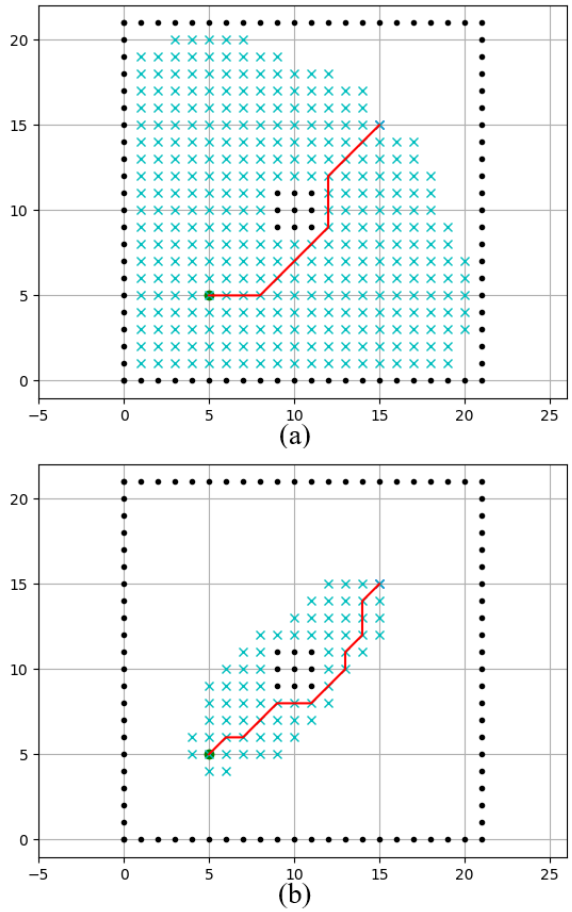
D. *Heuristic*

Penggunaan *heuristic* pada program utama yaitu pada bagian menentukan nilai c_id. *Heuristic* akan mengkalkulasikan antara *goal_position* dan *open_set*. Pada algoritma A* menggunakan *heuristic* sebagai acuan dalam menambahkan *node-node* baru. *Node* akan menuju langsung ke titik *goal* (bila tidak ada *obstacle*). Sub program *heuristic* dapat dilihat pada **Gambar 9**.

```
def heuristic (n1, n2):
    h = math.hypot(n1.x - n2.x, n1.y - n2.y)
    return h
```

Gambar 9. Sub Program *Heuristic*

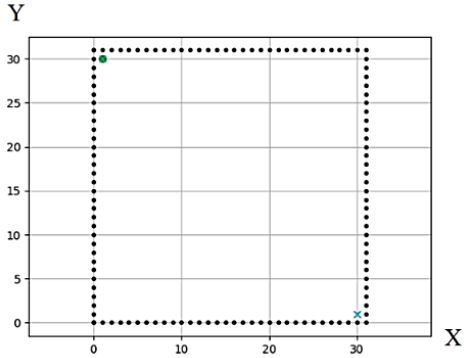
Pada program yang menggunakan *heuristic* menghasilkan *node* yang lebih sedikit. Berbeda dengan tidak menggunakan *heuristic* membutuhkan *node-node* yang lebih banyak untuk mencapai titik *goal*. Pada sub program *heuristic* menggunakan fungsi *math.hypot*. Fungsi dari *math.hypot* untuk menemukan jarak sisi miring dari segitiga siku–siku. Pada penggunaan *heuristic* akan membuat *node* bergerak langsung menuju titik *goal*. Bila tidak menggunakan *heuristic* maka *node* akan terus bertambah pada setiap arah, sehingga terjadi perbedaan jumlah *node* yang cukup jauh. Pada jalur yang dihasilkan dengan menggunakan *heuristic* atau tidak menggunakan *heuristic* menghasilkan jalur yang sama. Hasil jalur yang telah dibuat ditandai dengan garis yang berwarna merah. Perbedaan dari penggunaan *heuristic* dan tidak menggunakan *heuristic* dapat dilihat pada **Gambar 10**.



Gambar 10. Contoh Pengaplikasian (a) *Tanpa Heuristic* (b) *Heuristic*

E. *Grid Position*

Sub program ini digunakan untuk mengetahui suatu posisi pada koordinat *grid* berapa. Sub program position digunakan untuk menentukan posisi *start*, *goal* dan *node* berada pada posisi *grid*. Sub program ini digunakan pada sub program *verify*, *obstacle_map*, *final_path* serta *show animation*. Algoritma A* menggunakan disktritisasi *map* atau *grid map*. Untuk mengetahui suatu posisi pada koordinat *grid* yang ada pada koordinat xy, menggunakan sub program *position*. Contoh *map* dapat dilihat dari **Gambar 11**.



Gambar 11. Map

Berdasarkan *grid position* titik *start* berada pada *grid* 931 dan titik *goal* berada pada *grid* 61. *Index* merupakan nilai koordinat suatu *node* yang sudah dikonversikan ke koordinat *grid*. *Resolution* merupakan ukuran pencarian *node* sesuai dengan nilai *input resolution*. Sedangkan *min_position* merupakan nilai minimum dari tiap *position*. *Sub* program *grid position* dapat dilihat pada **Gambar 12**.

```
def position (self, index, min_position):
    pos = index * self.resolution + min_position
    return pos
```

Gambar 12. Sub Program *Grid Position*

F. XY *Index*

Sub program ini digunakan dalam menginisialisasi penentuan posisi dari *start_position* dan *goal_position*. Sub program XY *index* digunakan untuk menentukan posisi *start*, *goal* dan *node* berada pada koordinat xy berapa. Dimana *start_position* akan menjadi acuan mulainya pencarian terhadap *goal_position.*

Seperti pada **Gambar 11** posisi *start* berada pada koordinat x=1 dan y=30, sedangkan posisi *goal* berada pada koordinat x=30 dan y=1. Untuk mengetahui suatu posisi pada koordinat xy ada pada koordinat *grid* itu menggunakan sub program xy yang dapat dilihat pada **Gambar 13**. Pada sub program ini akan ditentukan setiap posisi dengan menggunakan koordinat X dan Y.

```
def xy (self, position, min_pos):
    return round((position - min_pos) /
                 self.resolution)
```

Gambar 13. Sub Program XY *Index*

G. Verify *Node*

Pada lingkungan yang digunakan terdapat area batas untuk proses pencarian titik tujuan. Terdapat area batas atas, bawah, kanan dan kiri. Proses pencarian akan terus menambahkan *node-node* dari titik *start* hingga titik *goal*. Pada proses pencarian akan dilakukan pencarian pada area yang ditentukan saja. Sehingga *node* tidak akan bertambah bila pada koordinat titik tersebut berada diluar area pencarian. Dengan mengabaikan titik koordinat yang berada diluar area pencarian maka pencarian tidak akan meluas sehingga dapat menjauhi titik tujuan. Pada proses pencarian akan menghindari *obstacle*, sehingga tidak akan menambahkan *node* pada titik yang terdapat *obstacle*. Pada sub program *verify node* akan mengidentifikasi dari calon posisi *node*. Setelah diketahui posisi calon *node* tersebut maka akan dibandingkan apakah titik tersebut berada pada area pencarian. Bila *node* tersebut berada diluar area pencarian maka tidak akan ditambahkan *node* pada titik tersebut. Sub program *verify node* dapat dilihat pada **Gambar 14**.

```
def verify (self, node):
    vx = self.position(node.x, self.min_ox)
    vy = self.position(node.y, self.min_oy)
    if vx < self.min_ox:
        return False
    elif vy < self.min_oy:
        return False
    elif vx >= self.max_ox:
        return False
    elif vy >= self.max_oy:
        return False
    if self.obstacle[node.x][node.y]:
        return False
    return True
```

Gambar 14. Sub Program *Verify Node*

Proses perbandingan akan dilihat pada area atas, bawah, kanan dan kiri. Pada area atas akan melihat apakah vy >= *self.max_oy*, pada area bawah akan melihat apakah vy < *self.min_oy*, pada area kanan akan melihat apakah vx >= *self.max_ox*, dan pada area kiri akan melihat apakah vx < *self.min_ox*. Bila titik koordinat tersebut berada diluar area pencarian maka akan *false*, sedangkan bila titik tersebut berada pada area pencarian maka akan *true*. Bila titik tersebut aman maka proses akan terus dilanjutkan pada tahap berikutnya.

H. *Obstacle* Map

Pada lingkungan pengujian yang digunakan terdapat beberapa *obstacle*. Pada proses pencarian jalur harus menghindari *obstacle*, serta area aman untuk pencarian. Sub program *obstacle map* dapat dilihat pada **Gambar 15**.

```
def obstacle_maps(self, obstacle_x, obstacle_y):
    self.min_ox=round(min(obstacle_x))
    self.min_oy=round(min(obstacle_y))
    self.max_ox=round(max(obstacle_x))
    self.max_oy=round(max(obstacle_y))
    self.width_x=round((self.max_ox - self.min_ox)/
                       self.resolution)
    self.width_y=round((self.max_oy - self.min_oy)/
                       self.resolution)

    self.obstacle=[[False for _ in range(self.width_y)]
                       for _ in range(self.width_x)]
    for ix in range(self.width_x):
        x=self.position(ix, self.min_ox)
        for iy in range(self.width_y):
            y=self.position(iy, self.min_oy)
            for iobstacle_x,iobstacle_y in zip(obstacle_x,
                                               obstacle_y):
                h = math.hypot(iobstacle_x - x,
                               iobstacle_y - y)
                if h <= self.radius:
                    self.obstacle[ix][iy] = True
                    break
```

Gambar 15. Sub Program *Obstacle* Map

Pada sub program ini diberikan nilai berdasarkan nilai dari inisialisasi *obstacle*. Dimana terdapat masukan dari *obstacle_x* dan *obstacle_y* kemudian diambil nilai minimum, sehingga menghasilkan nilai *min_ox*, *min_oy*, *max_ox* dan

max_oy. Pada *min_ox* merupakan batas dari *obstacle* area kiri, *min_oy* merupakan batas dari *obstacle* area bawah, *max_ox* merupakan batas dari *obstacle* area kanan, dan *max_oy* merupakan batas dari *obstacle* area atas. Kemudian dari nilai – nilai tersebut dikalkulasikan sehingga mendapatkan nilai *width_x dan width_y*. Pada *range width_x* dilakukan perhitungan untuk mendapatkan nilai x dengan menggunakan sub program *self.position.* Pada *range width_y* juga dilakukan perhitungan untuk mendapatkan nilai y dengan menggunakan sub program *self.position*. Kemudian dilakukan kalkulasi untuk menghasilkan nilai h berdasarkan *iobstacle_x* dan *iobstacle_y*. Dari nilai h tersebut maka akan dibandingkan dengan radius, apakah nilai h kurang sama dengan nilai radius. Jika nilai h kurang sama dengan nilai radius maka titik ix dan iy tersebut *true*. Radius tersebut merupakan jarak antara *node* dengan *obstacle*. Bila titik tersebut berada pada radius maka *node* tidak akan ditambahkan pada titik tersebut.

I. *Movement*

Pada hasil jalur yang didapatkan tentunya harus diketahui panjang dari jalur tersebut. Sehingga dengan mengetahui panjang jalur yang dihasilkan dapat diketahui jalur terdekat. Sub program untuk mengidentifikasi panjang jalur dapat dilihat pada **Gambar 16**.

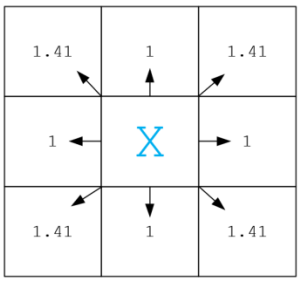
```
def movement ():
    movement = [[1, 0, 1],[0, 1, 1],
                [-1, 0, 1],[0, -1, 1],
                [-1, -1, math.sqrt(2)],
                [-1, 1, math.sqrt(2)],
                [1, -1, math.sqrt(2)],
                [1, 1, math.sqrt(2)]]
    return movement
```

Gambar 16. Sub Program *Movement*

Pada sub program *movement* terdapat *variable* berbentuk *array*. Terdapat delapan *variable*, yang masing-masing *variable* tersebut terdiri dari 3 kolom. Berikut pembahasan dari tiap baris dapat dilihat sebagai berikut:

- [1,0,1] pergerakan ke kanan dengan panjang 1
- [0,1,1] pergerakan ke atas dengan panjang 1
- [-1,0,1] pergerakan ke kiri dengan panjang 1
- [0,-1,1] pergerakan ke bawah dengan panjang 1
- [-1,-1,math.sqrt(2)] pergerakan diagonal (kiri, bawah) dengan panjang $\sqrt{2}$
- [-1,1,math.sqrt(2)] pergerakan diagonal (kiri, atas) dengan panjang $\sqrt{2}$
- [1,-1,math.sqrt(2)] pergerakan diagonal (kanan, bawah) dengan panjang $\sqrt{2}$
- [1,1,math.sqrt(2)] merupakan pergerakan diagonal (kanan, atas) dengan panjang $\sqrt{2}$

Panjang jalur tersebut akan menjadi acuan terhadap jarak dari tiap *node*. Setiap *node* akan bergerak ke delapan arah tersebut, maka akan memiliki jarak pada tiap pergerakan tersebut. Pada tiap *node* tersebut akan menciptakan jalur dari titik *start* ke *goal* dengan jarak yang telah didapatkan. Panjang jalur dapat dilihat pada **Gambar 17**.



Gambar 17. Panjang Jalur

J. *Final Path*

Sub program ini digunakan saat *node* telah mencapai titik *goal*. Setelah titik *goal* ditemukan maka program akan mengkalkulasikan jarak terendah dengan penggunakan sub program *final_path*.

Sehingga akan menghasilkan jalur dari titik *start* sampai titik *goal* dengan jarak yang paling dekat, berdasarkan *node-node* yang telah dibuat. Sub program *final path* dapat dilihat pada **Gambar 18**.

```
def final_path (self, goal_position, closed_set):
    sx, sy = [self.position(goal_position.x,
                self.min_ox)],[self.position(goal_position.y,
                                self.min_oy)]
    main = goal_position.main
    while main != -1:
        n = closed_set[main]
        sx.append(self.position(n.x, self.min_ox))
        sy.append(self.position(n.y, self.min_oy))
        main = n.main
    return sx, sy
```

Gambar 18. Sub Program *Final Path*

K. Inisialisasi Program

Input dari titik *start* dan *goal* digunakan untuk menentukan nilai dari *start_position* dan *goal_position*. Dimana *start_position* menjadi titik mulai *node* melakukan pencarian terhadap titik *goal*. Serta *goal_position* merupakan titik untuk *node* mengakhiri proses pencarian dari titik *goal*. Serta titik *goal* merupakan titik dimana tujuan dari proses pencarian jalur. Pada proses pencarian jalur harus dapat menghindari *obstacle* yang ada.

1) Inisialisasi Start dan Goal

Pada suatu program tentu saja terdapat suatu masukan agar program bekerja sesuai dengan harapan. Berikut *input* program dapat dilihat pada **Gambar 19**.

```
start_x = 5
start_y = 5
goal_x = 15
goal_y = 15
grid = 1
radius = 0
```

Gambar 19. Contoh Inisialisasi Program

Terdapat *start_x* dan *start_y* ini merupakan inisialisasi titik *start* pada program. Pada titik ini akan menjadi acuan terhadap pada koordinat yang akan menjadi awal pencarian jalur. Pada *goal_x* dan *goal_y* merupakan inisialisasi titik *goal*. Pada contoh diberikan titik *start* yaitu pada koordinat (5, 5). Sedangkan pada titik *goal* pada koordinat (15, 15). Pada titik yang nantinya akan menjadi koordinat akhir dari proses pencarian jalur. Pada proses pencarian jalur akan menghasilkan sebuah garis jalur. Garis jalur yang menghubungkan antara titik *start* dan titik *goal*. Pergerakan dari *node-node* dalam menemukan titik *goal* yaitu ditentukan pada inisisalisasi *grid*. Dimana pergerakan *node* akan bergerak tiap satu *grid*. Titik *start* ditandai dengan warna hijau sedangkan titik *goal* ditandai dengan tanda silang berwarna biru.

2) Inisialisasi Obstacle

Pada penelitian menggunakan *obstacle* agar mengetahui apakah program dapat bekerja dengan menghindari *obstacle*. Tentu dalam proses *path planning* tidak diperbolehkan untuk menabrak *obstacle*. Penggunaan *obstacle* dapat digunakan untuk melihat hasil dari proses *path planning*. Pada list program *for i in range* merupakan inisialisasi letak *obstacle*. Sebagai contoh *for i in range*(0, 21) berarti obstacle berada pada *range* 0 sampai 20. Pada *obstacle_x.append* dan *obstacle_y.append* merupakan inisialisasi letak dari *obstacle* tersebut. Sebagai contoh pada *obstacle_x.append*(i) dan *obstacle_y.append*(21) berarti *obstacle* berapa pada area kolom 21. Berikut contoh program *obstacle* dapat dilihat pada **Gambar 20**.

```
obstacle_x, obstacle_y = [], []
for i in range(0, 21):
    obstacle_x.append(i)
    obstacle_y.append(0)
for i in range(0, 21):
    obstacle_x.append(21)
    obstacle_y.append(i)
for i in range(0,22):
    obstacle_x.append(i)
    obstacle_y.append(21)
for i in range(0, 22):
    obstacle_x.append(0)
    obstacle_y.append(i)
for i in range(9, 12):
    obstacle_x.append(9)
    obstacle_y.append(i)
for i in range(9, 12):
    obstacle_x.append(10)
    obstacle_y.append(i)
for i in range(9, 12):
    obstacle_x.append(11)
    obstacle_y.append(i)
```

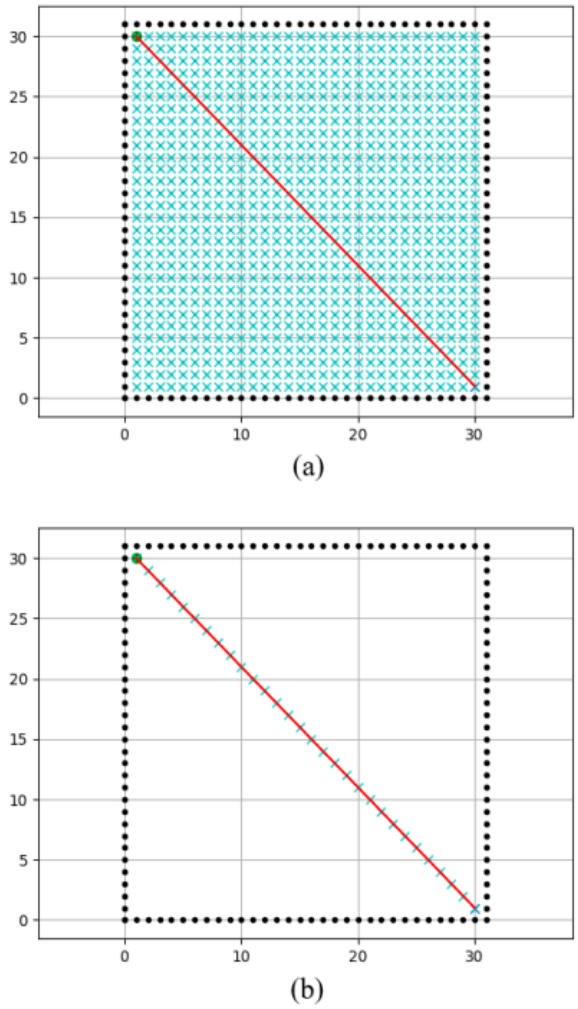
Gambar 20. Contoh Program *Obstacle*

III.HASIL DAN PEMBAHASAN

Pengujian yang dilakukan dengan membandingkan hasil algoritma BFS dan algoritma A*. Komparasi waktu komputasi diperlukan untuk mengetahui waktu yang diperlukan, sehingga menghasilkan jalur dari titik *start* hingga titik *goal*. Komparasi dilakukan pada lima skenario pengujian, yaitu pengujian pada lingkungan tanpa *obstacle*, *obstacle trap*, *obstacle* sederhana, *obstacle maze* dan *obstacle narrow*. Sehingga dengan beberapa perbandingan yang dilakukan untuk memastikan bahwa program berjalan sesuai dengan harapan. Pengujian dilakukan sebanyak 30 kali pada tiap lingkungan.

A. Lingkungan Pengujian Tanpa *Obstacle*

Pada pengujian waktu dilakukan untuk membandingkan antara algoritma BFS dan algoritma A*. Perbandingan waktu diperlukan untuk mengetahui waktu yang dibutuhkan untuk menghasilkan jalur. Hasil pengujian dapat dilihat pada **Gambar 21.**



Gambar 21. Lingkungan Pengujian Tanpa *Obstacle* (a) BFS (b) A*

Titik *start* berada pada titik koordinat (1, 30) sedangkan titik *goal* berada pada koordinat (30, 1). *Node* yang telah dibuat yaitu ditandai dengan silang berwarna biru, sedangkan untuk jalur yang telah dibuat ditandai dengan garis berwarna merah. Memiliki hasil jalur yang sama, dibuktikan dari nilai *path cost* yang sama yaitu 41,012. Seperti pada percobaan perancangan jalur yang telah dilakukan menghasilkan jarak jalur yang sama antara A* dan BFS [14]. Sehingga dengan hasil jalur yang sama membuktikan bahwa kinerja dalam perencanaan jalur tidak terjadi perbedaan. Berdasarkan perbandingan pada pengujian tanpa adanya *obstacle* menunjukan hasil yang berbeda, perbedaan jumlah *node* pada BFS lebih banyak dibandingkan dengan A*. Perbedaan jumlah *node* dikarenakan pada BFS hanya mempertimbangkan panjang langkah saja, sehingga penambahan *node* bertambah terus menerus pada sekitar *node* sebelumnya pada delapan arah. Berbeda dengan A* yang akan mendahulukan *node* yang jaraknya lebih dekat dengan titik *goal*. Sehingga pada A* menghasilkan *node* lebih efektif. Jumlah *node* yang telah dibuat pada BFS sebanyak 900 *node* dan pada A* sebanyak 30. Perbedaan *node* cukup jauh, bila menggunakan A* hanya menggunakan 3,33% dari jumlah *node* pada BFS. Perbedaan waktu eksekusi dapat dilihat pada **Tabel I**.

Tabel I. Waktu Eksekusi Lingkungan Tanpa *Obstacle*

No	Waktu Eksekusi		No	Waktu Eksekusi	
	BFS (detik)	A* (detik)		BFS (detik)	A* (detik)
1	6,83	1,33	**16**	6,42	1,30
2	6,70	1,33	**17**	6,31	1,30
3	6,79	1,31	**18**	6,33	1,30
4	6,85	1,36	**19**	6,30	1,29
5	6,85	1,30	**20**	6,22	1,29
6	6,30	1,34	**21**	6,42	1,35
7	6,60	1,29	**22**	6,24	1,29
8	6,30	1,28	**23**	6,32	1,29
9	6,30	1,27	**24**	6,37	1,29
10	6,28	1,29	**25**	6,35	1,34
11	6,29	1,29	**26**	6,27	1,35
12	6,32	1,35	**27**	6,39	1,28
13	6,32	1,36	**28**	6,62	1,35
14	6,48	1,35	**29**	6,51	1,35
15	6,33	1,29	**30**	6,39	1,30

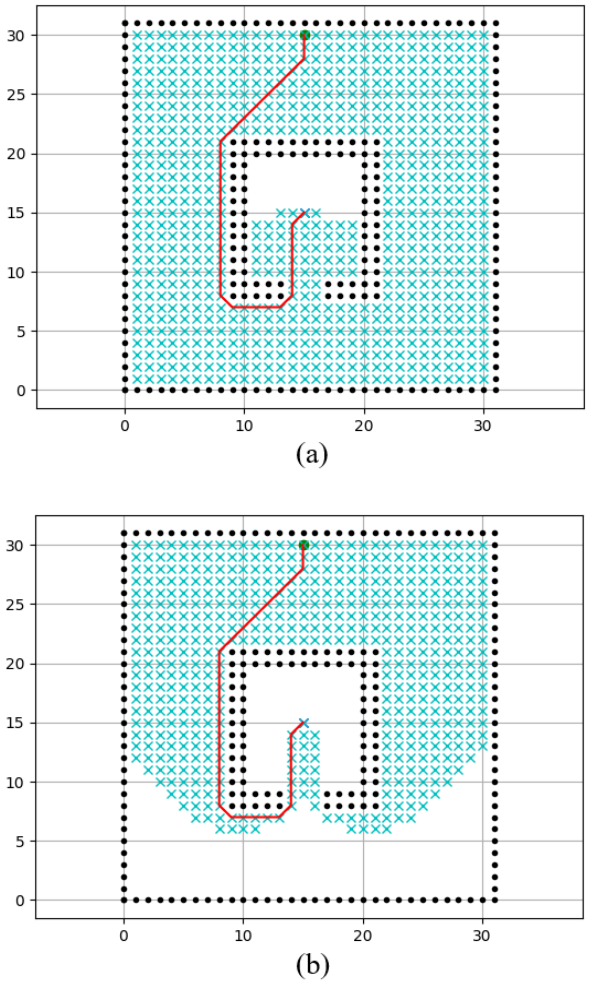
Perbedaan jumlah *node* tersebut mengakibatkan perbedaan waktu eksekusi. Waktu rata-rata yang dibutuhkan BFS untuk menghasilkan jalur yaitu 6,43 detik, sedangkan untuk A* membutuhkan waktu 1,33 detik. Serta waktu terlama pada BFS membutuhkan waktu yaitu 6,85 detik dan waktu tercepat yaitu 6,22 detik. Sedangkan pada algoritma A* menghasilkan waktu terlama 1,36 detik dan tercepat yaitu 1,27 detik. Keunggulan pada A* dimana hasil jalur yang sama tetapi waktu yang diperlukan lebih sedikit, sehingga A* lebih cepat dengan hasil yang sama. Statistik perbedaan waktu eksekusi dapat dilihat pada **Tabel II**.

Tabel II. Statistik Waktu Eksekusi Lingkungan Tanpa Obstacle

	Waktu Eksekusi	
	BFS	A*
Rata-rata	6,43 detik	1,31 detik
Terlama	6,85 detik	1,36 detik
Tercepat	6,22 detik	1,27 detik

B. Lingkungan Pengujian *Trap*

Pada pengujian ini menggunakan lingkungan pengujian perangkap. Hasil pengujian dapat dilihat pada **Gambar 22.**



Gambar 22. Lingkungan Pengujian *Trap* (a) BFS (b) A*

Pada lingkungan pengujian titik *start* yang berada pada koordinat (15, 30) serta titik *goal* berada pada koordinat (15, 15). Jalur yang dihasilkan melewati celah antara *obtacle*. Hasil dari perencanaan jalur yang dilakukan menghasilkan nilai *path cost* yang sama yaitu 39,14. Memiliki perbedaan pada jumlah *node* yang dibutuhkan yaitu pada BFS 773 *node* dan pada A* 528 *node*. Waktu eksekusi program dapat dilihat pada **Tabel III.**

Tabel III. Waktu Eksekusi Lingkungan *Trap*

No	Waktu Eksekusi		No	Waktu Eksekusi	
	BFS (detik)	A* (detik)		BFS (detik)	A* (detik)
1	5,59	3,67	**16**	5,33	3,79
2	5,34	3,60	**17**	5,34	3,62
3	5,61	3,69	**18**	5,40	3,58
4	5,50	3,69	**19**	5,59	3,58
5	5,36	3,65	**20**	5,33	3,58
6	5,37	3,60	**21**	5,36	3,58
7	5,42	3,59	**22**	5,60	3,57
8	5,33	3,62	**23**	5,32	3,57
9	5,33	3,58	**24**	5,33	3,57
10	5,61	3,55	**25**	5,35	3,56
11	5,36	3,57	**26**	5,37	3,63
12	5,37	3,52	**27**	5,43	3,59
13	5,36	3,59	**28**	5,35	3,56
14	5,52	3,54	**29**	5,52	3,64
15	5,38	3,61	**30**	5,53	3,55

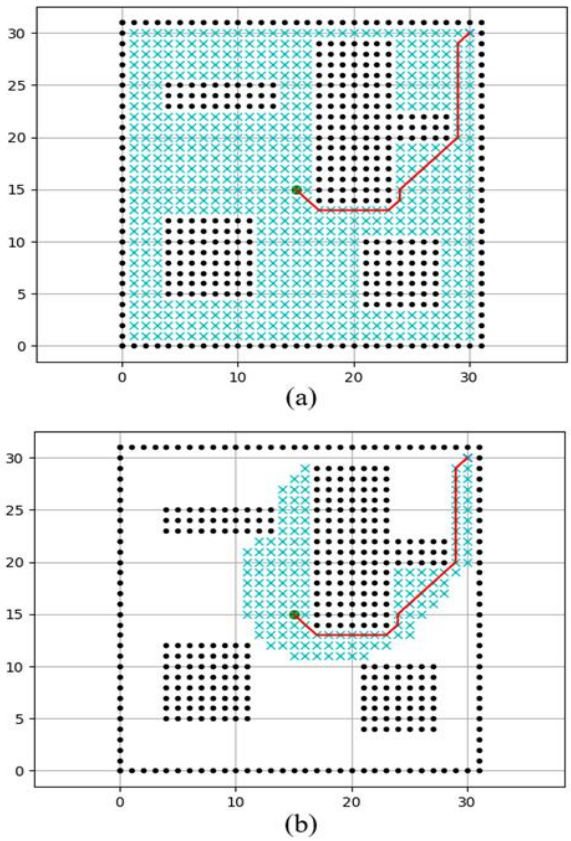
Waktu yang dibutuhkan untuk BFS yaitu rata-rata 5,42 detik dan A* yaitu 3,6 detik. Waktu tercepat pada BFS yaitu 5,32 detik sedangkan waktu terlama 5,61 detik. Pada A* dengan waktu tercepat 3,52 detik dan waktu terlama 3,79 detik. Statistik perbedaan waktu eksekusi dapat dilihat pada **Tabel IV**.

Tabel IV. Statistik Waktu Eksekusi Lingkungan *Trap*

	Waktu Eksekusi	
	BFS	A*
Rata-rata	5,42 detik	3,60 detik
Terlama	5,61 detik	3,79 detik
Tercepat	5,32 detik	3,52 detik

C. Lingkungan Pengujian *Obstacle* Sederhana

Pengujian dilakukan pada lingkungan yang terdapat 4 *obstacle* sederhana. Pengujian ini titik *start* berada pada koordinat (15, 15) dan untuk titik *goal* pada koordinat (30, 30). Hasil dapat dilihat pada **Gambar 23**.



Gambar 23. Lingkungan Pengujian *Obstacle* Sederhana (a) BFS (b) A*

Nilai *path cost* dari keduanya memiliki hasil yang sama yaitu 45,113. Waktu eksekusi dapat dilihat pada **Tabel V**.

Tabel V19. Waktu Eksekusi Lingkungan *Obstacle* Sederhana

No	Waktu Eksekusi		No	Waktu Eksekusi	
	BFS (detik)	A* (detik)		BFS (detik)	A* (detik)
1	4,64	2,25	**16**	4,45	2,15
2	4,60	2,15	**17**	4,30	2,13
3	4,56	2,17	**18**	4,35	2,10
4	4,54	2,15	**19**	4,28	2,16
5	4,57	2,22	**20**	4,32	2,08
6	4,31	2,16	**21**	4,41	2,12
7	4,30	2,10	**22**	4,31	2,09
8	4,33	2,18	**23**	4,64	2,12
9	4,36	2,09	**24**	4,29	2,14
10	4,36	2,09	**25**	4,31	2,08
11	4,45	2,11	**26**	4,36	2,11
12	4,66	2,12	**27**	4,41	2,11
13	4,35	2,12	**28**	4,34	2,12
14	4,37	2,10	**29**	4,60	2,14
15	4,31	2,12	**30**	4,31	2,14

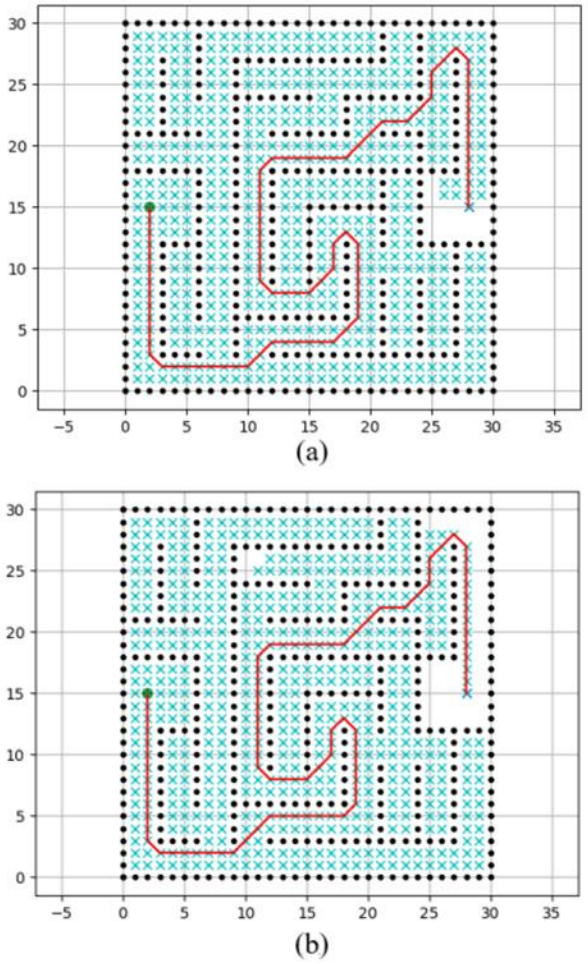
Pada pengujian ini jumlah *node* pada BFS yaitu 630 *node* sedangkan pada A* yaitu 148 *node*. Waktu yang dibutuhkan BFS yaitu dengan rata-rata 4,27 detik sedangkan A* yaitu 1,71. Pada pengujian ini BFS menghasilkan waktu tercepat dengan 4,17 detik dan waktu terlama 4,34 detik. Sedangkan pada pengujian A* menghasilkan waktu tercepat dengan 1,66 detik dan w.aktu terlama 1,79 detik. Statistik waktu yang dibutuhkan untuk mendapatkan hasil dapat dilihat pada **Tabel VI**.

Tabel VI. Statistik Waktu Eksekusi Lingkungan *Obstacle* Sederhana

	Waktu Eksekusi	
	BFS	**A***
Rata-rata	4,27 detik	1,71 detik
Terlama	4,43 detik	1,79 detik
Tercepat	4,17 detik	1,66 detik

D. Lingkungan Pengujian *Obstacle Maze*

Pada pengujian lingkungan pengujian ini menggunakan rintangan *maze*. Hasil pengujian dapat dilihat pada **Gambar 24**.



Gambar 24. Lingkungan Pengujian *Obstacle Maze* (a) BFS (b) A*

Titik *start* berada pada koordinat (2,15) dan titik *goal* berada pada titik koordinat (28, 15). Pengujian ini menghasilkan jalur yang sama dengan panjang jalur yaitu 92,87. Serta memiliki jumlah *node* yang hampir sama, dengan jumlah *node* pada pengujian BFS yaitu 588 *node* dan pada pengujian A* yaitu 562. Perbedaan *node* yang dibuat yaitu 26 *node*. Sehingga dengan menggunakan A* menghasilkan jumlah *node* yang lebih sedikit. Waktu pengujian lingkungan *maze* dapat dilihat pada **Tabel VII**.

Tabel VII. Waktu Eksekusi Lingkungan *Maze*

No	**Waktu Eksekusi**		**No**	**Waktu Eksekusi**	
	BFS (detik)	**A* (detik)**		**BFS (detik)**	**A* (detik)**
1	4,15	3,91	**16**	4,25	3,97
2	4,14	3,98	**17**	4,37	4,02
3	4,13	3,93	**18**	4,20	4,00
4	4,11	4,02	**19**	4,10	3,98
5	4,15	3,93	**20**	4,14	4,18
6	4,23	4,01	**21**	4,14	4,01
7	4,12	4,06	**22**	4,22	4,11
8	4,21	3,97	**23**	4,23	3,98
9	4,34	4,04	**24**	4,15	4,10
10	4,13	3,99	**25**	4,15	3,96
11	4,20	3,94	**26**	4,10	3,98
12	4,11	4,02	**27**	4,12	3,88
13	4,18	4,06	**28**	4,22	3,90
14	4,19	3,94	**29**	4,13	3,97
15	4,26	4,00	**30**	4,09	4,01

Hasil dari pengujian waktu yang dibutuhkan untuk BFS yaitu dengan rata-rata 4,18 detik dan pada pengujian A* membutuhkan waktu dengan rata-rata 4 detik. Waktu tercepat pada BFS yaitu 4,09 detik dan waktu terlama yaitu 4,37 detik. Sedangkan pada pengujian A* menghasilkan waktu tercepat yaitu 3,88 detik dan waktu terlama yaitu 4,18 detik. Seperti percobaan lingkungan *maze* yang telah dilakukan pada A* membutuhkan waktu yang lebih cepat dibandingkan dengan BFS, pada A* membutuhkan waktu 0.35 *millisecond* sedangkan BFS membutuhkan waktu 0.8 *millisecond* [15]. Statistik waktu pengujian dapat dilihat pada **Tabel VIII**.

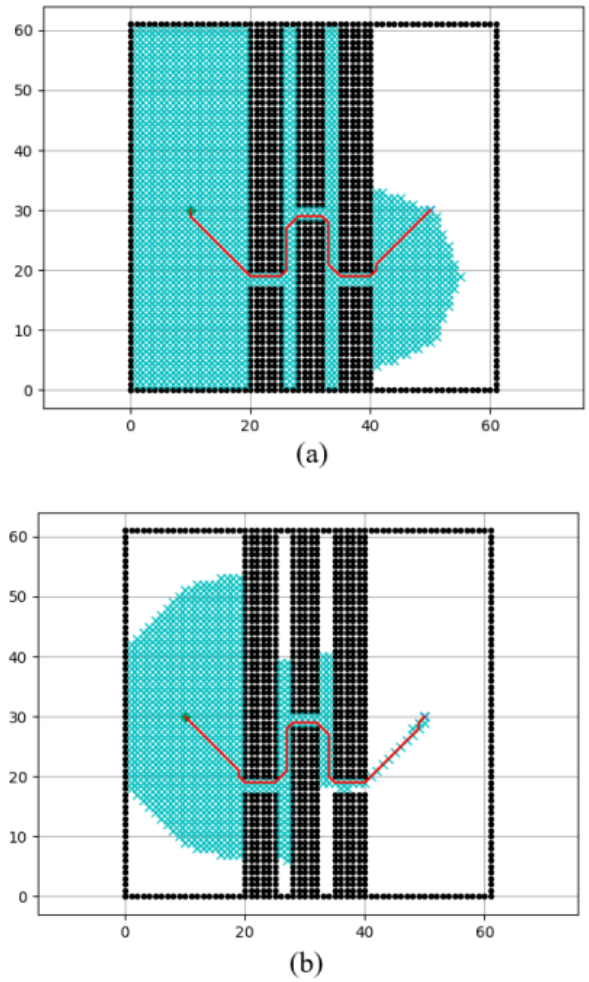
Tabel VIII. Statistik Waktu Eksekusi Lingkungan *Maze*

	Waktu Eksekusi	
	BFS	**A***
Rata-rata	4,18 detik	4 detik
Terlama	4,37 detik	4,18 detik
Tercepat	4,09 detik	3,88 detik

Perbedaan waktu tidak terlalu jauh karena *obstacle* yang tertutup atau hanya berbentuk jalur. Sehingga pada A* tetap melakukan penjelajahan ke area yang menjauhi titik *goal*. Pada A* tidak langsung menuju titik *goal* karena terhalang oleh *obstacle*, sehingga *node* baru akan dibuat pada titik yang tidak ada *obstacle*. Dengan begitu maka *node* baru akan mengikuti jalur pada rintangan *maze*. Setelah mendekati titik *goal* dan tidak ada rintangan, maka *node* baru pada A* langsung menuju titik *goal*.

E. Lingkungan Pengujian *Obstacle Narrow*

Pada pengujian ini menggunakan obstacle yang sempit, kemudian pada sisi *start* dan *goal* memiliki area tanpa *obstacle* yang cukup luas. Terdapat 3 garis yang masing masing berjarak 2 *node*. Sehingga pada pengujian pertama ini menghasilkan perbedaan waktu yang cukup jauh, dibandingkan dari pengujian sebelumnya. Titik *start* terdapat pada koordinat (10, 30) dan titik *goal* pada koordinat (50, 30). Hasil pencarian jalur dapat dilihat pada **Gambar 25**.



Gambar 25. Lingkungan Pengujian *Obstacle Narrow* 1 (a) BFS (b) A*

Node bergerak melewati celah dari obstacle tersebut. Pada area titik *start* terdapat area tanpa *obstacle* cukup luas. Perbedaan pada area titik *start* dimana pada BFS seluruh area telah terjelajahi, serta pada area celah pada *obstacle* juga telah terjelajahi. Pada saat *node* melewati garis *obstacle* terakhir pada A*, *node* baru langsung menuju titik *goal*. Sedangkan pada BFS *node* baru tetap bergerak sesuai dengan panjang langkangnya saja. Sehingga pada BFS *node* baru setelah garis *obstacle* akhir membuat seperti lingkaran. Perbedaan waktu eksekusi pada lingkungan *narrow* dapat dilihat pada **Tabel IX**.

Tabel IX. Waktu Eksekusi Lingkungan *Narrow* 1

No	Waktu Eksekusi		No	Waktu Eksekusi	
	BFS (detik)	A* (detik)		BFS (detik)	A* (detik)
1	17,02	8,08	**16**	17,74	7,60
2	17,57	7,82	**17**	16,97	7,39
3	18,00	7,67	**18**	18,13	7,27
4	17,09	7,94	**19**	17,07	7,42
5	16,90	7,64	**20**	16,93	7,33
6	16,91	6,91	**21**	17,43	7,32
7	17,27	8,14	**22**	16,75	7,40
8	16,42	8,25	**23**	16,94	7,20
9	16,80	7,61	**24**	17,01	7,27
10	17,34	7,29	**25**	17,12	7,05
11	16,83	7,56	**26**	17,10	7,34
12	16,96	7,12	**27**	17,13	7,10
13	16,95	8,93	**28**	17,03	7,22
14	17,50	7,42	**29**	18,15	7,28
15	16,96	7,38	**30**	17,06	7,18

Perbandingan jumlah *node* yang dibutuhkan pada BFS yaitu 1730 *node*, sedangkan pada A* membutuhkan 905 *node*. Garis jalur yang dihasilkan didapatkan sama dengan panjang 66,77 *node* pada kedua pengujian. Statistik waktu pengujian dapat dilihat pada **Tabel X**.

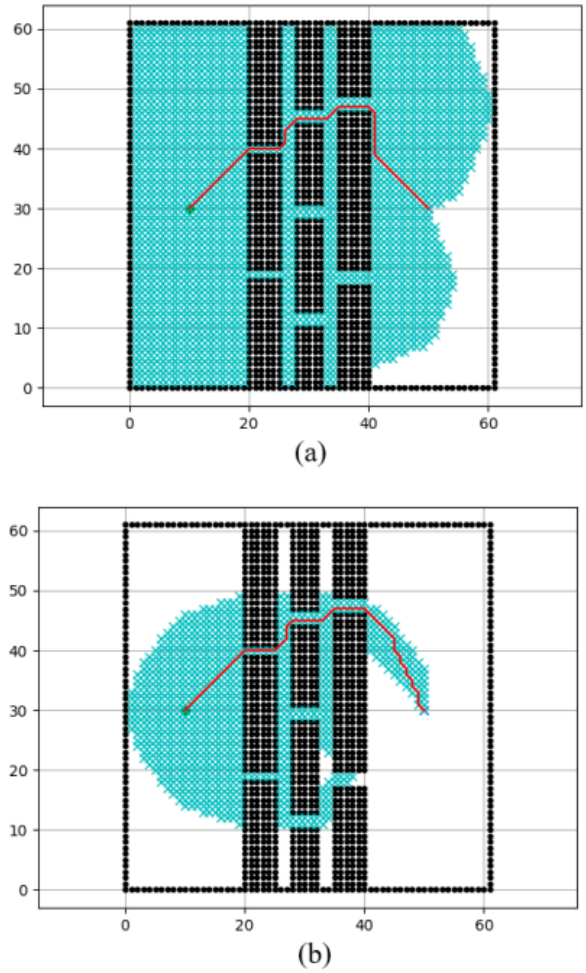
Tabel X. Statistik Waktu Eksekusi Lingkungan *Narrow* 1

	Waktu Eksekusi	
	BFS	A*
Rata-rata	17,17 detik	7,50 detik
Terlama	18,15 detik	8,93 detik
Tercepat	16,42 detik	6,91 detik

Hasil dari waktu untuk pengujian BFS yaitu dengan rata-rata 17,17 detik dan pada pengujian A* yaitu dengan rata-rata 7,5 detik. Pada BFS memerlukan waktu terlama yaitu 18,15 detik dan

waktu tercepat yaitu 16,42 detik. Sedangkan pada A* memerlukan waktu terlama yaitu 8,93 detik dan waktu tercepat yaitu 6,91 detik. Dari hasil pengujian menghasilkan waktu yang cukup jauh. Sehingga dengan membutuhkan waktu yang lebih lama maka program tidak efektif.

Pada percobaan kedua lingkungan *narrow* dilakukan menambahkan celah lain berdasarkan lingkungan *narrow* sebelumnya. Dengan menambahkan celah lain maka berdasarkan pengujian menghasilkan jalur yang berbeda dengan percobaan sebelumnya. Hasil dari pengujian dapat dilihat pada **Gambar 26**.



Gambar 26. Lingkungan Pengujian *Obstacle Narrow* 2 (a) BFS (b) A*

Panjang jalur yang dihasilkan dari algoritma BFS maupun A* menghasilkan panjang yang sama yaitu 59,36. Pada percobaan lingkungan *narrow* kedua menghasilkan jarak yang lebih dekat dibandingkan dengan pengujian sebelumnya. Menghasilkan jarak jalur yang lebih dekat karena jalur melewati celah yang baru. Hal tersebut membuktikan bahwa pada proses *path planning* dengan menggunakan algoritma BFS dan A* akan menghasilkan jarak yang terdekat. Pada lingkungan *narrow* kedua menghasilkan perbedaan yang cukup jauh. Waktu pengujian lingkungan *narrow* kedua dapat dilihat pada **Tabel XI**.

Tabel XI. *Waktu Eksekusi Lingkungan Narrow 2*

No	Waktu Eksekusi		No	Waktu Eksekusi	
	BFS (detik)	A* (detik)		BFS (detik)	A* (detik)
1	26,04	6,62	**16**	26,58	6,48
2	26,46	6,50	**17**	25,68	6,53
3	26,03	6,57	**18**	25,89	6,51
4	26,12	6,58	**19**	26,37	6,71
5	27,80	6,59	**20**	27,17	6,51
6	26,13	6,53	**21**	26,07	6,40
7	25,95	6,82	**22**	26,54	6,46
8	26,24	6,82	**23**	26,65	6,48
9	26,48	6,57	**24**	26,58	6,53
10	25,78	6,59	**25**	26,59	6,52
11	25,94	6,65	**26**	26,64	6,53
12	26,25	6,56	**27**	26,59	6,90
13	26,27	6,74	**28**	26,58	6,48
14	26,17	6,48	**29**	25,63	6,51
15	26,18	6,45	**30**	26,44	6,51

Pada percobaan lingkungan *narrow* kedua menghasilkan rancangan jarak yang lebih pendek dibandingkan dengan percobaan pertama tetapi waktu yang dibutuhkan BFS lebih lama. Statistik waktu pengujian dapat dilihat pada **Tabel XII**.

Tabel XII. *Statistik Waktu Eksekusi Lingkungan Narrow 2*

	Waktu Eksekusi	
	BFS	A*
Rata-rata	26,33 detik	6,57 detik
Terlama	27,8 detik	6,9 detik
Tercepat	25,63 detik	6,4 detik

Waktu yang dibutuhkan pada pengujian lingkungan *narrow* kedua pada algoritma BFS membutuhkan waktu yang lebih lama yaitu dengan rata- rata 26,33 detik. Sedangkan pada A* waktu yang dibutuhkan yaitu dengan rata – rata 6.57 detik. Waktu tercepat pada A* yaitu 6,4 detik sedangkan pada BFS yaitu 25.63 detik. Serta waktu terlama pada A* yaitu 6.9 detik dan pada BFS yaitu 27.8 detik.

F. Analisa Jumlah *Node*

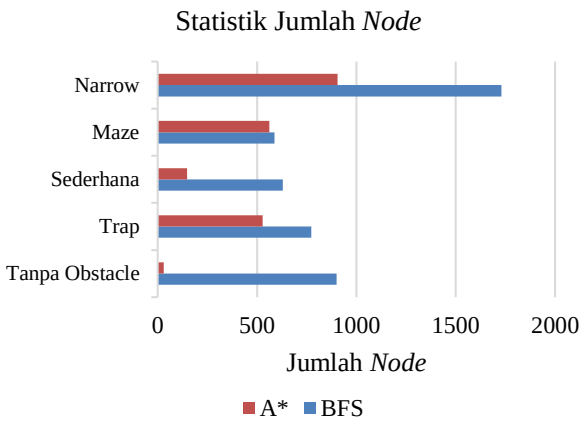
Pada pencarian jalur diperlukannya *node-node* dalam melakukan proses pencarian jalur. *Node-node* akan terus bertambah dari titik *start* hingga mencapai titik *goal*. Jumlah *node* pada tiap

lingkungan terjadi perbedaan antara algoritma BFS dan A*. Perbandingan jumlah *node* tiap lingkungan dapat dilihat pada **Tabel XIII**.

Tabel XIII20. Jumlah *Node*

Lingkungan Pengujian	Jumlah *Node*		Perbedaan Jumlah *Node* (BFS – A*)
	BFS	A*	
Tanpa *Obstacle*	900	30	870
Trap	773	528	245
Sederhana	630	148	482
Maze	588	562	26
Narrow	1730	905	825

Seperti pengujian pada lingkungan tanpa *obstacle* menghasilkan perbedaan yang cukup jauh dengan perbedaan 870 *node*, pada pengujian lingkungan *trap* menghasilkan perbedaan 245 *node*, pada pengujian lingkungan sederhana menghasilkan perbedaan 482 *node*, pada lingkungan *maze* menghasilkan perbedaan 26 *node*, dan pada lingkungan *narrow* menghasilkan perbedaan 825 *node*. Algoritma A* membutuhkan *node* lebih sedikit dibandingkan dengan BFS pada semua lingkungan. Grafik statistik jumlah *node* dapat dilihat pada **Gambar 27**.



Gambar 27. Grafik Jumlah *Node*

Seperti pada percobaan yang telah dilakukan pada lingkungan *maze* dengan membandingkan A*, dijikstra dan BFS menghasilkan jumlah *node* yang berbeda, yang dimana jumlah *node* pada A* lebih sedikit dibandingkan djikstra dan BFS [14][15]. Pada lima lingkungan pengujian algoritma BFS memerlukan *node* lebih banyak dibandingkan dengan A*. Pada lingkungan pengujian *maze* terjadi perbedaan yang tidak terlalu jauh. Berbeda dengan lingkungan tanpa *obstacle* perbedaan jumlah *node* yang jauh. Karena pada A* *node* langsung menuju titik *goal* tanpa menambahkan ke setiap tetangga *node* yang telah dibuat. Sedangkan pada BFS dilingkungan tanpa *obstacle node* bertambah pada setiap tetangga *node* tersebut. Sehingga dilingkungan tanpa *obstacle* menghasilkan jumlah perbedaan *node* yang cukup jauh.

IV.KESIMPULAN

Dari pengujian perencanaan jalur pada lima lingkungan, bahwa algoritma A* membutuhkan waktu yang lebih cepat dibandingkan BFS. Tetapi pada pengujian pada lingkungan pengujian *maze* terjadi perbedaan waktu yang sedikit. Pada BFS memerlukan waktu tercepat 4,09 detik serta pada A* memerlukan waktu tercepat 3,88 detik. Pada perbedaan yang tidak terlalu jauh algoritma A* masih unggul. Hal ini terjadi karena algoritma BFS hanya mempertimbangkan panjang langkah saja. Sedangkan pada algoritma A* mempertimbangkan jarak ke tujuan. Perbedaan jumlah *node* yang telah dibuat oleh algoritma A* dan BFS menghasilkan perbedaan. Pada algoritma A* membutuhkan jumlah *node* lebih sedikit dibandingkan dengan algoritma BFS. Pada pengujian lingkungan tanpa *obstacle* menghasilkan perbedaan yang cukup jauh. Pada BFS dilingkungan tanpa *obstacle* membutuhkan 900 *node* sedangkan pada A* membutuhkan 30 *node*. Hal itu terjadi karena pada A* *node* langsung menuju titik *goal* tanpa menambahkan ke setiap tetangga *node* yang telah dibuat. Sedangkan pada BFS *node* bertambah pada setiap tetangga *node* tersebut. Berbeda dengan lingkungan *maze* yang memiliki perbedaan jumlah *node* cukup sedikit yaitu 26 *node*. Hal ini terjadi karena lingkungan pengujian yang berbentuk seperti jalur. Sehingga *node* baru pada A* akan mengikuti jalur tersebut. Perbedaan terjadi saat mendekati titik *goal*, karena tidak ada *obstacle* maka *node* baru A* langsung menuju titik *goal*.

Pada percobaan yang telah dilakukan menghasilkan bahwa algoritma A* merupakan metode yang lebih optimal dibandingkan dengan algoritma BFS. Sehingga untuk mengetahui apakah algoritma A* merupakan perancangan jalur yang paling optimal maka dapat dilakukan perbandingan dengan algoritma sejenis lainnya. Serta hasil *path planning* dapat diaplikasikan seperti pada robot.

DAFTAR PUSTAKA

[1] A. Budiati dan P. I. Santosa, "Implementasi Algoritma Best Path Planning Untuk Pencarian Rute Trans Jogja," *Semin. Nas. Inform. SEMNASIF*, vol. 1, hlm. 1, 2015.

[2] P. Ida, "Rancang Bangun Robot Pengangkut Barang Dengan Teknik Path Planning," *Elektron. Telekomun. Comput.*, vol. 14, hlm. 10, 2019.

[3] C. Liu, S. Zhang, dan A. Akbar, "Ground Feature Oriented Path Planning for Unmanned Aerial Vehicle Mapping," *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.*, vol. 12, no. 4, hlm. 1175–1187, Apr 2019.

[4] B. Fu *dkk.*, "An Improved A* Algorithm For The Industrial Robot Path Planning With High Success Rate And Short Length," *Robot. Auton. Syst.*, vol. 106, hlm. 26–37, Agu 2018.

[5] D. Drake, S. Koziol, dan E. Chabot, "Mobile Robot Path Planning with a Moving Goal," *IEEE Access*, vol. 6, hlm. 12800–12814, 2018.

[6] M. Aria, "Algoritma Perencanaan Jalur Kendaraan Otonom berbasis Hibridisasi Algoritma BFS dan Path Smoothing," *Telekontran J. Ilm. Telekomun. Kendali Dan Elektron. Terap.*, vol. 8, no. 1, hlm. 13–22, Jun 2020.

[7] J. Wang, W. Chi, C. Li, C. Wang, dan M. Q.-H. Meng, "Neural RRT*: Learning-Based Optimal Path Planning," *IEEE Trans. Autom. Sci. Eng.*, vol. 17, no. 4, hlm. 1748–1758, Okt 2020.

[8] Y. Gao, J. Liu, M. Q. Hu, H. Xu, K. P. Li, dan H. Hu, "A New Path Evaluation Method for Path Planning with Localizability," *IEEE Access*, vol. 7, hlm. 162583–162597, 2019.

[9] A. K. Guruji, H. Agarwal, dan D. K. Parsediya, "Time-efficient A* Algorithm for Robot Path Planning," *Procedia Technol.*, vol. 23, hlm. 144–149, 2016.

[10] T. Oral dan F. Polat, "MOD* Lite: An Incremental Path Planning Algorithm Taking Care of Multiple Objectives," *IEEE Trans. Cybern.*, vol. 46, no. 1, hlm. 245–257, 2016.

[11] F. Duchoň *dkk.*, "Path Planning with Modified a Star Algorithm for a Mobile Robot," *Procedia Eng.*, vol. 96, hlm. 59–69, 2014.

[12] M. B. Subramanian, D. K. Sudhagar, dan G. RajaRajeswari, "Intelligent Path Planning of Mobile Robot Agent by Using Breadth First Search Algorithm," *Int. J. Innov. Res. Sci. Eng. Technol.*, vol. 3, no. 3, hlm. 5, 2014.

[13] P. Gao, Z. Liu, Z. Wu, dan D. Wang, "A Global Path Planning Algorithm for Robots Using Reinforcement Learning," *2019 IEEE Int. Conf. Robot. Biomim. ROBIO*, hlm. 1693–1698, 2019.

[14] M. B. Subramanian, D. K. Sudhagar, dan G. RajaRajeswari, "Optimal Path Forecasting of an Autonomous Mobile Robot Agent Using Breadth First Search Algorithm," *Int. J. Mech. Mechatron. Eng.*, vol. 14, no. 2, hlm. 85–89, 2014.

[15] S. D. H. Permana, K. B. Y. Bintoro, B. Arifitama, dan A. Syahputra, "Comparative Analysis of Pathfinding Algorithms A *, Dijkstra, and BFS on Maze Runner Game," *IJISTECH Int. J. Inf. Syst. Technol.*, vol. 1, no. 2, hlm. 1, Mei 2018.

[16] S. M. M. R. Al-Arif, A. H. M. I. Ferdous, dan S. H. Nijami, "Comparative Study of Different Path Planning Algorithms: A Water based Rescue System," *Int. J. Comput. Appl.*, vol. 39, no. 5, hlm. 25–29, Feb 2012.