Perbandingan Berbagai Implementasi Algoritma Pencarian Jalan pada Permainan Berbasis Petak

Muhammad Rizky Ismail Faizal 13518148
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13518148@std.stei.itb.ac.id

Pencarian jalan adalah suatu komponen yang penting dalam sangat banyak permainan. Dibutuhkan suatu algoritma pencarian jalan yang mangkus dan sangkil untuk dapat membuat komponen-komponen penting dalam permainan seperti kecerdasan buatan, sistem pergerakan pemain, dan komponen lainnya. Terdapat dua algoritma yang sangat dikenal dalam bidang ini, yaitu algoritma pencarian Dijkstra dan algoritma pencarian A* (A-Star).

perbandingan, pencarian jalan, graf, dijkstra, a-star.

I. PENDAHULUAN

Pada permainan berbasis petak, terdapat dua tipe sistem pergerakan yang umum digunakan: sistem pergerakan satu langkah dan sistem pergerakan banyak langkah. Pada sistem pergerakan satu langkah, pemain menekan tombol pergerakan dan karakter yang dikendalikan oleh pemain bergerak ke arah yang diinginkan. Sistem seperti ini relatif sederhana, dan tidak memerlukan sistem pencarian jalan.

Sementara itu, pada sistem pergerakan banyak langkah, pemain memilih petak mana yang ingin dijadikan tujuan, kemudian menekan tombol pergerakan. Hal yang terjadi setelahnya adalah permainan akan menentukan jalan yang paling cepat dan tepat (tepat dengan maksud dapat dilalui) dari titik awal posisi karakter menuju titik yang dipilih oleh pemain.

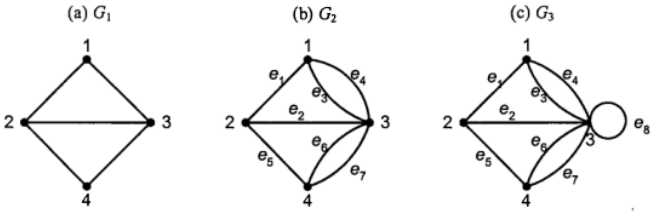
Untuk dapat menentukan jalan yang paling cepat dan tepat tersebut, umumnya diperlukan suatu sistem tersendiri. Sistem ini adalah sistem pencarian jalan yang terdiri dari struktur data dan algoritma untuk menentukan jalan. Salah satu algoritma yang paling dikenal dalam sistem jenis ini adalah algoritma Dijkstra, sementara algoritma A-star adalah salah satu pengembangan dari algoritma Dijkstra.

Pada makalah ini akan dibandingkan kemangkusan dan kesangkilan dari berbagai implementasi kedua algoritma tersebut.

II. LANDASAN TEORI

A. Graf

Secara matematis, graf didefinisikan sebagai pasangan himpunan (V, E), ditulis dengan notasi G = (V,E), yang dalam hal ini V adalah himpunan tidak-kosong dari simpul-simpul (*vertices* atau *node*) dan E adalah himpunan sisi (*edges* atau *arcs*) yang menghubungkan sepasang simpul. [1]



Gambar 2.1 Tiga buah graf (a) graf sederhana, (b) graf ganda, dan (c) graf semu. [1]

Pada implementasi dari suatu graf, diperlukan fungsi dan prosedur yang dapat menentukan keterhubungan antar simpul dengan berbagai cara. Contohnya dengan membuat suatu fungsi *neighbors* yang akan mengembalikan simpul simpul yang terhubung dengan suatu simpul masukan.

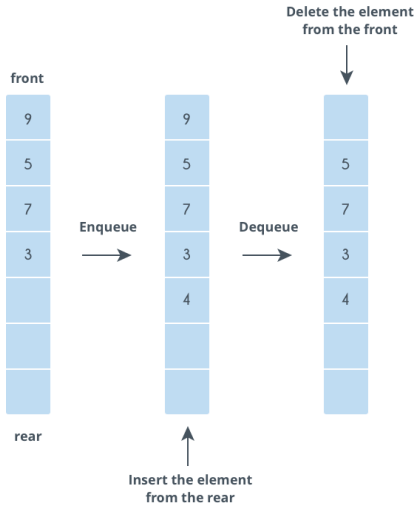
Graf juga terbagi menjadi beberapa jenis. Salah satu jenis graf adalah graf berbobot dimana tiap sisi memiliki bobot yang ditentukan. Graf berbobot biasa digunakan untuk merepresentasikan jaringan, baik dalam pembangunan jalan maupun untuk *networking*. Pada graf berbobot, salah satu masalah populer adalah bagaimana mencari jalur dengan bobot atau biaya terrendah dari suatu simpul ke suatu simpul lainnya.

B. Queue

Queue adalah struktur data yang mengikuti pola *First In First Out* (FIFO). Artinya elemen pertama yang ditambahkan ke dalam queue akan menjadi elemen pertama yang akan dihapus.

Elemen akan ditambahkan dari belakang dan akan dihapus dari depan. *Queue* dapat diumpamakan seperti antrean suatu wahana di taman bermain. Orang yang berada di posisi terdepan akan menjadi yang pertama masuk ke dalam wahana. [2]

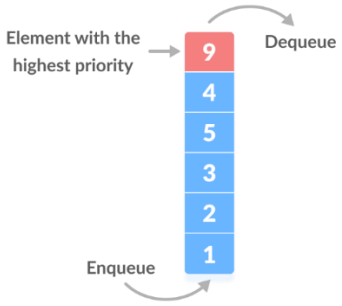
Yang membedakan *queue* dengan struktur data lainnya adalah cara mengakses elemennya dalam implementasi. Dalam implementasi, umumnya hanya elemen pertama (paling depan) dalam sebuah *queue* yang dapat diakses dan/atau dikeluarkan, dan elemen hanya bisa dimasukan di belakang elemen terakhir. Ini bisa berbeda pada beberapa tipe turunan *queue* lainnya.



Gambar 2.2 Contoh *queue*. [2]

C. Priority Queue

Sebuah *priority queue* adalah suatu tipe spesial dari *queue* dimana tiap elemennya diasosiasikan dengan sebuah prioritas dan disajikan terurut berdasarkan prioritas tersebut. [3]



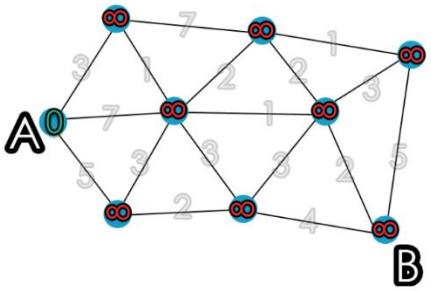
Gambar 2.3 Contoh *priority queue*. [3]

Priority queue dapat diterapkan dengan mengambil sebuah penerapan *queue* biasa dan kemudian mengubah struktur data elemennya dan fungsi penambahan elemen atau fungsi pengambilan elemennya. Struktur data tiap elemennya diwajibkan memiliki suatu variabel yang dapat dibandingkan satu sama lain untuk menjadi tolak ukur keberurutan *queue* tersebut.

Fitur keberurutan *priority queue* dapat diterapkan dengan mengubah fungsi penambahan elemen sehingga elemen yang baru disisipkan di lokasi yang tepat berdasarkan variabel prioritasnya, atau dengan mengubah fungsi pengambilan elemen sehingga elemen yang diambil selalu elemen dengan prioritas terrendah.

D. Algoritma Dijkstra

Algoritma Dijkstra mencari jalur tersingkat dari suatu titik menuju titik akhir dengan membuat suatu himpunan simpul yang memiliki jarak minimum dari titik sumber. [4]



Gambar 2.4 Contoh graf berbobot dimana dapat dicari jalur terdekat dari A menuju B [4]

Dalam algoritma ini, diambil suatu simpul awal pada graf, kemudian dilihat semua simpul yang terhubung dengan simpul yang sedang diambil: tiap simpul yang terhubung dimasukkan kedalam suatu *priority queue* dengan prioritas berdasarkan biaya yang diperlukan untuk mencapai simpul terhubung tersebut ditambah biaya simpul yang sedang diambil. Setelah semua simpul yang terhubung dimasukkan kedalam *queue*, simpul yang sedang diambil dimasukkan kedalam himpunan simpul yang sudah diproses, kemudian proses diulang pada simpul teratas pada queue (yaitu simpul dengan jarak terdekat dari simpul permulaan). Proses pencarian selesai ketika ditemukan bahwa simpul pertama pada *queue* adalah simpul akhir.

E. Algoritma A-Star

Algoritma A-Star adalah suatu algoritma yang dikembangkan dari algoritma Dijkstra. Cara kerjanya hampir sama dengan algoritma Dijkstra biasa, hanya saja ditambahkan satu faktor lain yang berupa variabel heuristis dalam pengurutan simpul didalam *queue*.

7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18	19	20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

Gambar 2.5 Contoh jalur terdekat dari petak biru ke petak hijau. [5]

Algoritma A-Star sering dibilang seperti algoritma Dijkstra "berarah" karena umumnya variabel heuristis yang dipertimbangkan adalah jarak dari simpul akhir ke suatu simpul. Ini diilustrasikan pada gambar 2.5: angka yang ada pada tiap petak adalah jarak tiap petak dengan simpul hijau. Semakin dekat suatu petak dengan simpul hijau, semakin kecil nilainya.

Implementasi pemertimbangan faktor lain ini dapat dilakukan dengan mengurutkan setiap elemen pada *priority queue*

berdasarkan nilai yang berupa hasil penjumlahan dari biaya yang diperlukan untuk mencapai simpul itu dengan variabel heuristis.

III. Implementasi Algoritma

Dalam pengujian ini, dibuat suatu aplikasi yang dapat menguji kedua algoritma. Aplikasi dibuat dengan menggunakan bahasa pemrograman Python, dengan banyaknya baris hampir mencapai empat ratus baris. Karena empat ratus baris terlalu banyak untuk sepenuhnya dicantumkan disini, hanya akan dicantumkan bagian-bagian dari kode yang paling penting dan relevan. Kode selengkapnya dapat diakses pada tautan berikut: https://github.com/rzkyif/gamma.

A. Variabel Pengujian

```
. . .

DIJKSTRA = False
OPTIMAL = True
FIXED_COST = False

. . .
```

Gambar 3.1 Variabel pengujian dalam kode.

Pada program yang dibuat, digunakan beberapa variabel pengujian yang dapat diatur saat program dijalankan supaya dapat ditemukan implementasi mana yang paling mangkus dan sangkil.

Variabel pengujian pertama, yaitu DIJKSTRA, menandakan apakah akan digunakan algoritma pencarian Dijkstra atau tidak. Jika variabel ini bernilai True, akan digunakan algoritma pencarian Dijkstra (kasus ini ditandai label Dijkstra). Jika bernilai False, akan digunakan algoritma pencarian A-star (kasus ini ditandai label A-star).

Kedua, terdapat variabel OPTIMAL, yang menandakan kondisi berhenti fungsi pencarian. Ketika variabel OPTIMAL bernilai True, pencarian akan dihentikan hanya ketika simpul akhir menempati posisi pertama pada *queue* (kasus ini ditandai label Optimal). Ini menandakan sudah tidak mungkin ada jalur yang lebih singkat menuju posisi akhir dari posisi awal. Ketika bernilai False, pencarian akan dihentikan ketika simpul akhir pertama ditemukan (kasus ini ditandai label First).

Pengubahan variabel OPTIMAL menjadi True, secara teori, akan membuat pencarian lebih mangkus dalam mencari jalur yang paling singkat. Metode ini juga adalah metode yang seharusnya diimplementasikan berdasarkan teori. Akan tetapi, dengan mengubah variabel ini menjadi False, jalur akan ditemukan dengan lebih cepat meskipun terdapat resiko jalur bukanlah yang paling singkat.

Terakhir, terdapat variabel FIXED_COST, yang menandakan metode yang digunakan untuk mendapatkan biaya yang diperlukan untuk mencapai suatu posisi. Ketika variabel ini bernilai True, akan digunakan perhitungan jarak euclidean dari posisi awal ke posisi yang bersangkutan untuk menentukan biaya (kasus ini ditandai label Fixed Cost). Ketika bernilai False, akan digunakan sistem pendahulu dimana harga dari tiap simpul didapat dari harga simpul yang dilalui untuk mencapai simpul tersebut ditambah satu langkah yang diperlukan untuk mencapai simpul tersebut (kasus ini ditandai label Generated Cost).

Implementasi yang sesuai dengan teori adalah dengan variabel FIXED_COST bernilai False. Ini memastikan jalur yang didapat benar benar jalur yang terdekat. Akan tetapi, dengan mengubah variabel FIXED_COST menjadi True, terdapat kemungkinan jalur akan didapat dengan lebih cepat dengan resiko jalur bukanlah yang paling singkat.

B. Graf

```
class AStarGraph:

  . . .

  def neighbors(self, node):
    return [node.pos.up(), node.pos.down(), node.pos.righ
t(), node.pos.left()]

  def weight(self, a, b):
    return 1
```

Gambar 3.2 Implementasi graf dalam kode.

Dalam memeriksa keterhubungan antar posisi, digunakan graf. Pada permainan berbasis petak, keterhubungan antarpetak tidak perlu direpresentasikan dalam sebuah graf karena umumnya jelas bahwa tiap petak yang bersebelahan dihitung terhubung, terutama pada implementasi ini. Karena itu struktur data graf "AStarGraph" yang dibuat diatas hanya digunakan untuk menyimpan fungsi sederhana untuk mencari simpul-simpul yang bertetangga dengan suatu simpul dan fungsi untuk mencari biaya melalui suatu sisi yang pada implementasi ini selalu bernilai 1.

C. Simpul

```
class AStarNode:
  def __init__(self, pos, parent, start, end):
    self.pos = pos
    self.parent = parent
    self.cost = self.generate_cost(start)
    self.nearness = self.generate_nearness(end)
    self.priority = self.cost + self.nearness

  . . .

  def generate_cost(self, start):
    if FIXED_COST:
      return math.sqrt((self.pos.i-start.i)*(self.pos.i-
start.i) + (self.pos.j-start.j)*(self.pos.j-start.j))
```

```
    else:
      if self.parent:
        return self.parent.cost + 1
      else:
        return 0

  def generate_nearness(self, end):
    return math.sqrt((end.i-self.pos.i)*(end.i-
self.pos.i) + (end.j-self.pos.j)*(end.j-self.pos.j))
```

Gambar 3.3 Implementasi simpul dalam kode.

Tiap simpul pada graf direpresentasikan dengan suatu struktur data "AStarNode" yang menyimpan informasi mengenai posisi simpul tersebut di peta permainan (pos), simpul yang dilalui untuk mencapai simpul tersebut (parent), biaya yang dibutuhkan untuk mencapai posisi simpul tersebut (cost), jarak posisi simpul tersebut dengan posisi simpul akhir (nearness), dan prioritas simpul tersebut (priority).

Bagian kode yang patut diperhatikan adalah bagian fungsi generate_cost yang memperhitungkan biaya yang dibutuhkan untuk mencapai posisi simpul. Dengan variabel pengujian FIXED_COST, biaya akan berupa nilai jarak euclidean posisi simpul dengan posisi simpul awal. Tanpa variabel pengujian tersebut, biaya akan berupa biaya simpul yang dilalui untuk mencapai simpul tersebut ditambah 1 (langkah yang diperlukan untuk mencapai simpul tersebut).

Selain itu, terdapat juga fungsi generate_nearness yang digunakan untuk memperhitungkan jarak posisi simpul tersebut dengan posisi simpul akhir. Fungsi ini tidak terikat parameter uji seperti fungsi generate_nearness, karena heuristis yang digunakan untuk A-star pada semua implementasi adalah jarak euclidean posisi tiap simpul dengan posisi simpul akhir. Dapat dilihat juga bahwa prioritas simpul hanyalah hasil penjumlahan dari biaya simpul tersebut (cost) dan jarak posisi simpul tersebut dengan posisi simpul akhir (nearness).

D. Priority Queue

```
class AStarQueue:

  . . .

  def Insert(self, node):
    if self.ContainBetter(node):
      return
    i = 0
    while i < len(self.queue):
      if DIJKSTRA:
        cmp1 = self.queue[i].cost
        cmp2 = node.cost
      else:
        cmp1 = self.queue[i].priority
        cmp2 = node.priority
      if cmp1 > cmp2:
        break
      else:
        i += 1
    if i >= len(self.queue):
      self.queue.append(node)
    else:
      self.queue.insert(i, node)

  def ContainBetter(self, node):
    return any(x.pos == node.pos and x.cost <= node.cost
for x in self.queue)

  . . .
```

Gambar 3.4 Implementasi *priority queue* dalam kode.

Dalam proses penjalanan algoritma pencarian jalan, setiap node yang ditemukan dimasukkan kedalam suatu *priority queue* yang memiliki struktur data "AStarQueue" di atas. Struktur data ini menyimpan informasi mengenai simpul yang sudah ditemukan di peta, terurut membesar sesuai suatu parameter.

Pengurutan elemen pada implementasi *priority queue* ini dilakukan pada saat pemasukan elemen di fungsi Insert. Pada fungsi ini, diperiksa variabel pengujian DIJKSTRA: jika ada, elemen yang dimasukkan diurutkan berdasarkan biaya saja (cost); jika tidak, elemen yang dimasukkan diurutkan berdasarkan prioritas elemen (priority).

Pada implementasi ini, fungsi Insert memeriksa apakah elemen yang ingin dimasukkan pada *queue* sudah ada atau belum di dalam *queue*. Jika elemen sudah ada, diperiksa apakah biaya (cost) elemen yang ada di dalam *queue* lebih besar atau tidak. Jika lebih besar, elemen yang ingin dimasukkan menggantikan elemen yang lebih dulu ada dalam *queue*. Proses ini memanfaatkan fungsi ContainBetter.

E. Algoritma Pencarian Jalan

```
def astar(self, start=None, end=None):
  if not start:
    start = self.playerpos
  if not end:
    end = self.cursorpos
  self.trace = [[None for j in range(self.width)] for i i
n range(self.height)]
  self.checked = 0
  self.pathed = 0
  finishset = []
  nodeset = []
  queue = AStarQueue()
  graph = AStarGraph()
  currentnode = AStarNode(start, None, start, end)
  nodeset.append(currentnode)
```

```
  found = False
  while not found:
    for pos in graph.neighbors(currentnode):
      if not self.valid(pos):
        continue
      node = AStarNode(pos, currentnode, start, end)
      i = next((i for i, x in enumerate(nodeset) if (x.po
s == node.pos and x.cost > node.cost)), None)
      if i:
        nodeset.insert(i, node)
      elif not any(x.pos == node.pos for x in nodeset):
          nodeset.append(node)
      if pos in finishset:
        continue
      queue.Insert(node)
      if not OPTIMAL:
        if node.pos == end:
          found = True
          break
      if node.pos != end:
        self.mark_checked(pos)
    finishset.append(currentnode.pos)
    currentnode = queue.Pop()
    if OPTIMAL:
      if currentnode.pos == end:
        found = True
        break
  if found:
    pos = end
    while pos != start:
      pos = next(node.parent.pos for node in nodeset if n
ode.pos == pos)
      if pos != start:
        self.mark_path(pos)
  time.sleep(1)
  self.stop = True
```

Gambar 3.5 Implementasi fungsi pencarian jalan dalam kode.

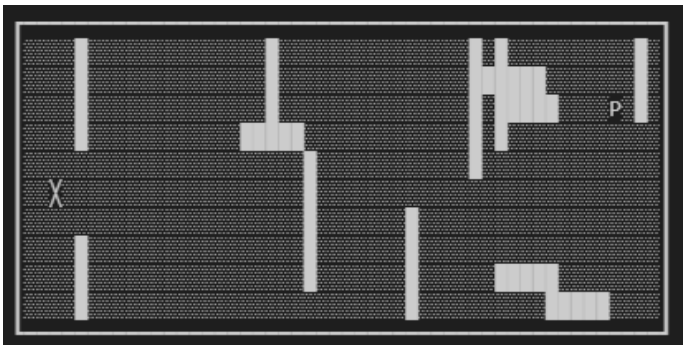
Fungsi pencarian yang sama digunakan untuk algoritma Dijkstra dan algoritma A-star. Ini karena perbedaan fundamental di antara keduanya ada pada sistem pengurutan pada *queue* yang sudah dijelaskan sebelumnya. Fungsi ini secara kasar melakukan implementasi langsung dari algoritma A-star pada landasan teori, hanya saja dengan beberapa perubahan dan penyesuaian untuk dapat bekerja lebih mangkus dan sangkil.

Pertama, dibuat suatu simpul baru untuk posisi awal. Setelah itu dibuat simpul baru untuk setiap posisi yang terhubung dengan posisi awal tersebut. Pada setiap pembuatan simpul baru, jika simpul tersebut belum pernah diperiksa, simpul tersebut akan dimasukkan kedalam *queue*. Jika setiap simpul di sekitar posisi awal sudah dibuat dan dimasukkan kedalam *queue*, elemen teratas pada *queue* diambil dan proses pemeriksaan yang sebelumnya dilakukan pada posisi awal diulang.

Perlu diperhatikan juga pada bagian pemeriksaan variabel pengujian OPTIMAL. Jika ditemukan variabel pengujian OPTIMAL, pencarian akan dihentikan hanya ketika simpul akhir menempati posisi pertama pada *queue*. Jika tidak, pencarian akan dihentikan tepat ketika simpul akhir ditemukan dalam pencarian.

IV. PENGUJIAN

Pertama, semua implementasi diujikan dengan peta permainan yang sama yaitu peta permainan berikut.

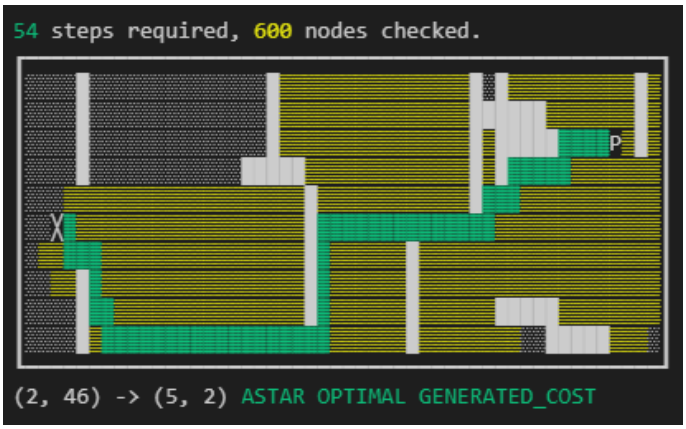


Gambar 4.1 Peta permainan.

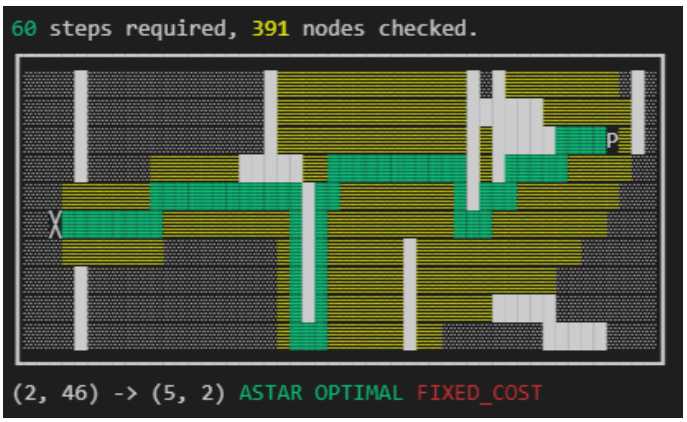
Penjelasan peta permainan:

1. P, yaitu posisi pemain sekaligus posisi simpul awal
2. X, yaitu posisi *cursor* sekaligus posisi simpul akhir
3. Petak abu, yaitu daerah yang dapat dilalui pemain
4. Petak putih, yaitu dinding yang tidak dapat dilalui pemain

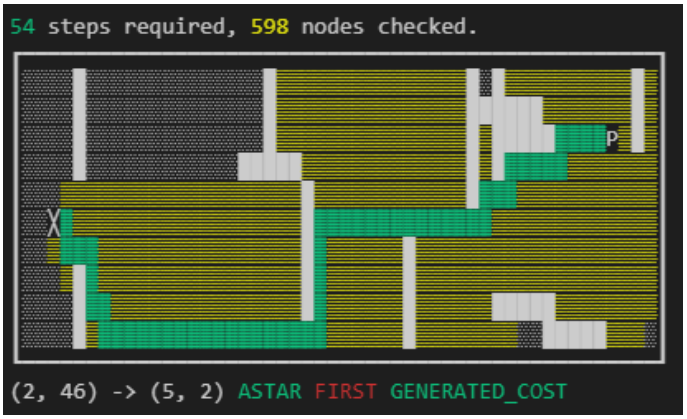
Hasil dari pengujian dengan menggunakan tiap implementasi sebagai berikut.



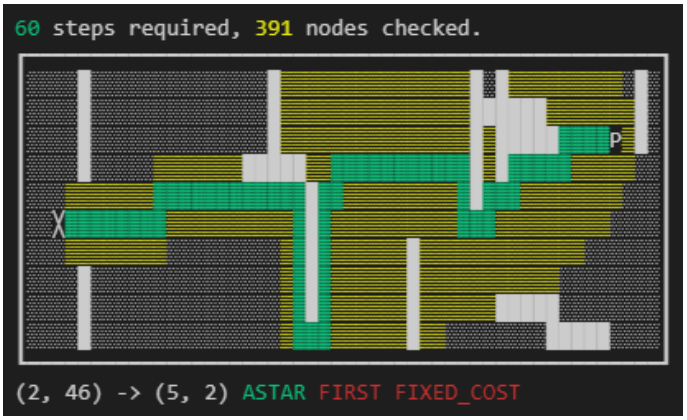
Gambar 4.2 Jalur yang ditemukan algoritma A-star (optimal, generated cost).



Gambar 4.3 Jalur yang ditemukan algoritma A-star (optimal, fixed cost).



Gambar 4.4 Jalur yang ditemukan algoritma A-star (first, generated cost).

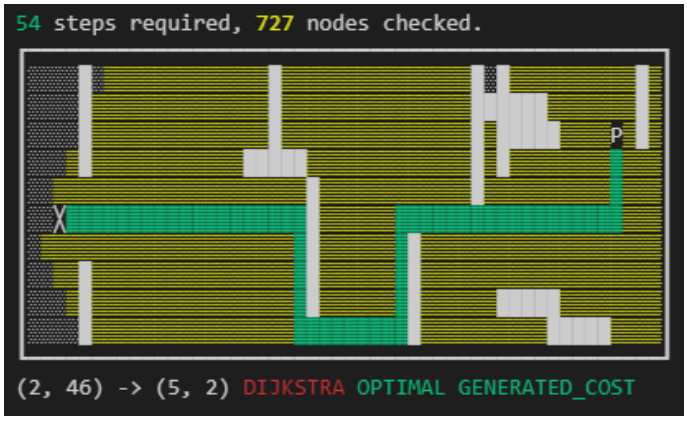


Gambar 4.5 Jalur yang ditemukan algoritma A-star (first, fixed cost).

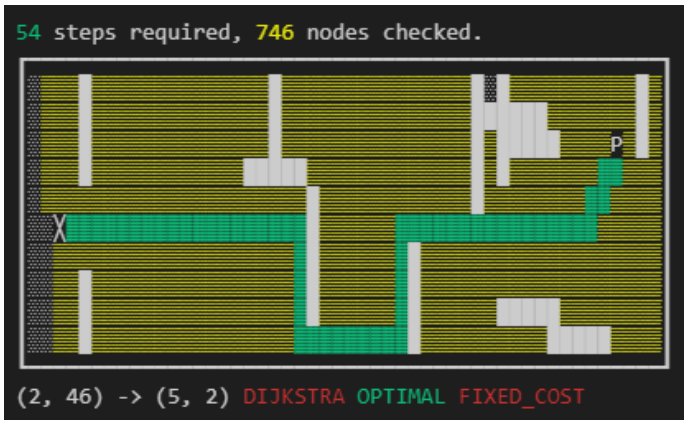
Dapat dilihat pada gambar 4.2, 4.3, 4.4 dan 4.5 hasil jalur yang ditemukan empat implementasi algoritma A-star yang berbeda.

Warna kuning menandai daerah yang sempat diuji oleh algoritma saat mencari jalur menuju simpul akhir. Dapat dilihat bahwa dengan variabel pengujian FIXED_COST, algoritma menjadi sangat sangkil dan hanya menguji hampir setengah dari banyak petak yang diuji implementasi lainnya.

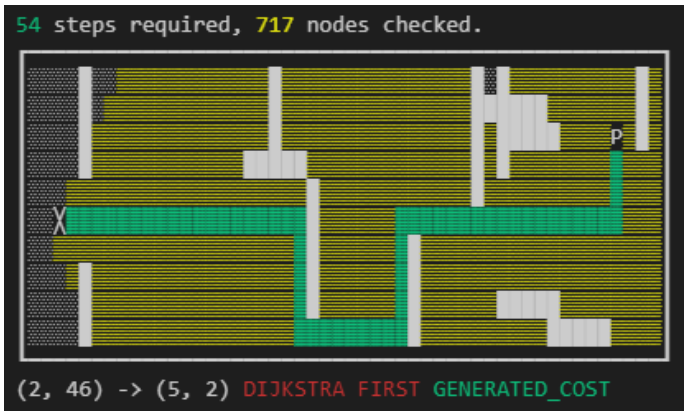
Warna hijau menandakan jalur yang dipilih algoritma. Dapat dibilang implementasi ini tidak mangkus karena banyak langkah yang diperlukan dalam melalui jalur hasil implementasi lain lebih sedikit dibandingkan pada implementasi dengan pengaktifan variabel FIXED_COST.



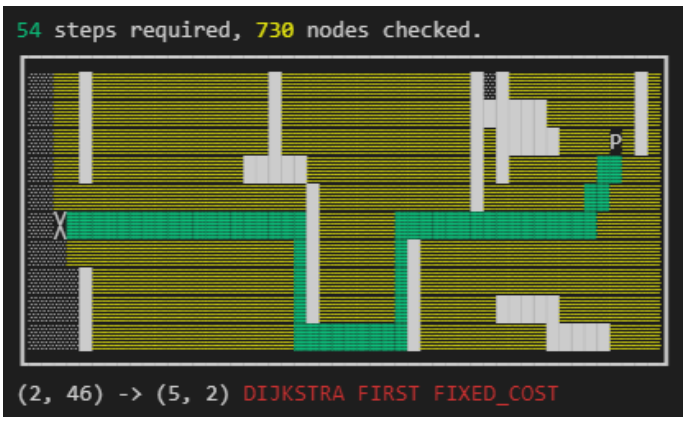
Gambar 4.6 Jalur yang ditemukan algoritma Dijkstra (optimal, generated cost).



Gambar 4.7 Jalur yang ditemukan algoritma Dijkstra (optimal, fixed cost).



Gambar 4.8 Jalur yang ditemukan algoritma Dijkstra (first, generated cost).



Gambar 4.9 Jalur yang ditemukan algoritma Dijkstra (first, fixed cost).

Pada gambar 4.6, 4.7, 4.8, dan 4.9 ditampilkan hasil jalur yang ditemukan oleh empat implementasi algoritma Dijkstra yang berbeda.

Dapat dilihat secara sekilas bahwa algoritma A-star jauh lebih sangkil dibandingkan algoritma Dijkstra dalam menemukan jalur, karena semua implementasi algoritma Dijkstra memeriksa hampir semua petak sebelum menemukan jalan yang tepat.

Akan tetapi jika dilihat dari kemangkusan, dapat dilihat bahwa semua implementasi algoritma Dijkstra menghasilkan jalur yang hanya memerlukan 54 langkah sementara dua dari empat implementasi algoritma A-star menghasilkan jalur yang memerlukan 60 langkah.

Setelah tiap implementasi diuji, dilakukan pengujian dengan lima buah peta permainan berbeda supaya dapat diperoleh nilai pada tabel berikut.

	A-Star, Optimal, Generated Cost			A-Star, Optimal, Fixed Cost			A-Star, First, Generated Cost		
No	Jalur	Cek	%	Jalur	Cek	%	Jalur	Cek	%
1.	34	278	12.23	42	185	22.7	34	277	12.27
2.	8	27	29.63	8	21	38.1	8	25	32
3.	11	80	13.75	11	44	25	11	78	14.1
4.	53	326	16.26	55	241	22.82	53	326	16.26
5.	31	308	10.06	31	95	32.63	31	307	10.1
	Rata-rata		16.386	Rata-rata		28.25	Rata-rata		16.946

Tabel 4.1 Data uji dalam lima peta permainan berbeda.

	A-Star, First, Fixed Cost			Dijkstra, Optimal, Generated Cost			Dijkstra, Optimal, Fixed Cost		
No	Jalur	Cek	%	Jalur	Cek	%	Jalur	Cek	%
1.	42	185	22.7	34	696	4.885	38	705	5.39
2.	8	21	38.1	8	230	3.478	8	250	3.2
3.	11	44	25	11	156	7.051	11	105	10.48
4.	55	241	22.82	53	643	8.243	53	671	7.899
5.	31	95	32.63	31	675	4.593	33	675	4.889
	Rata-rata		28.25	Rata-rata		5.65	Rata-rata		6.3716

Tabel 4.2 Data uji dalam lima peta permainan berbeda.

	Dijkstra, First, Generated Cost			Dijkstra, First, Fixed Cost		
No	Jalur	Cek	%	Jalur	Cek	%
1.	34	683	4.978	38	692	5.491
2.	8	201	3.98	8	221	3.62
3.	11	144	7.639	11	81	13.58
4.	53	629	8.426	53	657	8.067
5.	31	653	4.747	33	664	4.97
		Rata-rata	5.954		Rata-rata	7.1456

Tabel 4.3 Data uji dalam lima peta permainan berbeda.

Penjelasan tabel:

1. Jalur, yaitu berapa banyak langkah dalam jalur yang ditemukan algoritma dari posisi simpul awal ke posisi simpul akhir. Lebih kecil lebih baik.
2. Cek, yaitu berapa banyak petak yang diperiksa oleh algoritma sebelum menemukan jalur yang diperhitungkan terbaik. Lebih kecil lebih baik.
3. %, yaitu persentase jalur terhadap cek pada algoritma tersebut. Lebih besar lebih baik.
4. Rata-rata, yaitu rata-rata persentase pada algoritma tersebut. Lebih besar lebih baik.
5. Warna hijau menandakan nilai terbaik pada suatu nomor pengujian.
6. Warna kuning menandakan nilai diantara terbaik dan terburuk pada suatu nomor pengujian
7. Warna merah menandakan nilai terburuk pada suatu nomor pengujian.

V. Analisis

Dengan melihat hasil pengujian secara grafis dan secara nilai pada tabel, dapat dilihat suatu pola pada performa antar implementasi.

Dengan pengamatan grafis pada gambar 4.2 hingga gambar 4.9, dapat dilihat dengan jelas bahwa secara kesangkilan algoritma A-star jauh lebih sangkil dibandingkan dengan algoritma Dijkstra. Hasil ini sesuai teori dan memang pada dasarnya algoritma A-star dibuat untuk membuat algoritma Dijkstra lebih sangkil.

Akan tetapi, dari segi kemangkusan, algoritma Dijkstra ternyata dapat menjadi sama mangkusnya atau bahkan lebih mangkus dibandingkan algoritma A-star, tergantung pada implementasi yang digunakan.

Dapat dilihat pada tabel 4.1 hingga tabel 4.3 bahwa dalam lima pengujian dengan memanfaatkan peta permainan acak, algoritma Dijkstra tidak pernah unggul ataupun setara dengan algoritma A-star pada kolom banyaknya pengecekan yang dilakukan (tidak ada warna hijau pada bagian Dijkstra). Akan tetapi, dapat dilihat juga bahwa pada kolom langkah yang

dibutuhkan untuk melalui jalur terdapat cukup banyak kasus kesetaraan antara implementasi dengan algoritma Dijkstra dengan implementasi dengan algoritma A-star.

Terdapat juga kejadian sesuai hipotesis yang dapat dilihat, bahwa ternyata implementasi yang paling sangkil adalah implementasi A-star dengan label Fixed Cost dan implementasi tersebut mengalami beberapa kegagalan dalam menentukan jalur yang paling singkat.

Dapat dilihat juga bahwa ternyata dampak dari variabel FIXED_COST tidak terlalu besar dalam meningkatkan kesangkilan suatu implementasi. Bahkan adanya variabel tersebut membuat salah satu pengujian implementasi Dijkstra menjadi gagal dalam menemukan jalur paling singkat.

Terakhir, dapat terlihat bahwa pengubahan variabel OPTIMAL sama sekali tidak mempengaruhi hasil banyaknya langkah yang diperlukan dalam jalur pada pengujian semua implementasi pada tabel. Secara teori, seharusnya ada kasus dimana pengujian dengan label First menemukan jalur yang tidak sesingkat yang ditemukan pengujian dengan label Optimal, seperti pada pengujian di gambar 4.2 dan 4.3. Akan tetapi ternyata dalam lima kali pengujian tidak ditemukan kasus yang serupa.

VI. KESIMPULAN

Dari pengamatan hasil uji dan analisis yang sudah dilakukan dapat disimpulkan bahwa ternyata kemangkusan algoritma Dijkstra dapat setara dengan kemangkusan algoritma A-star, tetapi kesangkilan algoritma A-star jauh mengalahkan kesangkilan algoritma Dijkstra.

Dapat juga disimpulkan bahwa terdapat beberapa pengubahan yang dapat dilakukan pada implementasi algoritma Dijkstra dan algoritma A-star untuk membuat algoritma tersebut menjadi lebih sangkil. Akan tetapi perubahan-perubahan ini meningkatkan kesangkilan dengan biaya kemangkusan: kadang jalur yang dihasilkan jadi bukan jalur yang tersingkat. Perubahan-perubahan ini dapat memiliki dampak yang minim dengan biaya yang minim seperti pengubahan variabel OPTIMAL pada pengujian dan dampak yang lumayan besar seperti pengubahan variabel FIXED_COST pada pengujian.

VII. UCAPAN TERIMA KASIH

Pertama-tama penulis bersyukur atas berkat dan rahmat Allah SWT. karena dengan kehendak-Nya penulis dimudahkan dalam menyelesaikan makalah ini dengan baik. Penulis juga mengucapkan terima kasih kepada orang tua, sahabat, dan teman-teman penulis yang senantiasa memberi dukungan, motivasi, dan kasih sayang selama proses penulisan makalah ini. Penulis turut mengucapkan terima kasih kepada dosen Matematika Diskrit, Bapak Rinaldi Munir, yang telah memberikan pelajaran tentang Aljabar Boolean di mata kuliah Matematika Diskrit ini.

REFERENSI

[1] Munir, Rinaldi. Matematika Diskrit. Bandung: Penerbit Informatika. 2010.
[2] https://indonesia.hackerearth.com/konsep-dasar-queue/ diakses 09:53, 6 Desember 2019.
[3] https://www.programiz.com/dsa/priority-queue diakses 10:00, 6 Desember 2019.
[4] https://brilliant.org/wiki/dijkstras-short-path-finder/ diakses 10:04, 6 Desember 2019.
[5] https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2 diakses 10:09, 6 Desember 2019.

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, 6 Desember 2019

Muhammad Rizky Ismail F. 13518148