BAB II

DASAR TEORI

2.1 *Rapidly-exploring Random Tree* (RRT)

Algoritma yang digunakan pada RRT dapat dilihat pada **Gambar 2.1.** terlihat beberapa proses yang harus dilakukan yaitu *RandomSample, NearestNeighbor, Steer, dan InsertNode.* Setiap iterasi, node q_{rand} diambil secara acak dari ruang pencarian. Kemudian q_{rand} *akan digunakan pada NearestNeighbor* untuk mencari node pada pohon pencarian yang lebih dekat dengan q_{rand}. Node terdekat ini dinamakan $q_{nearest}$. Selantjutnya akan dibangun node baru antara $q_{nearest}$ dan q_{rand}. Node yang baru terbentuk ini dinamakan q_{new}. Jarak antara $q_{nearest}$ dengan q_{new} adalah sejauh Δq dari $q_{nearest}$ jika diantara $q_{nearest}$ dan q_{new} tidak ada hambatan maka q_{new} akan ditambahkan ke pohon pencarian.

Algorithm 1 : $T = (V, E) \leftarrow RRT(q_{init})$

1. $T \leftarrow InitializeTree()$
2. $T \leftarrow InsertNode(\emptyset, q_{init}, T)$
3. **for** $k \leftarrow 1$ **to** N **do**
4. $\quad q_{rand} \leftarrow RandomSample(k)$
5. $\quad q_{nearest} \leftarrow NearestNeighbor(q_{rand}, Q_{near}, T)$
6. $\quad q_{new} \leftarrow Steer\ (q_{nearest}, q_{rand}, \Delta q)$
7. $\quad$ **if** $Obstaclefree(q_{new}, q_{nearest})$ **then**
8. $\quad\quad T \leftarrow InsertNode(q_{parent}, q_{new}, T)$
9. **end**

Gambar 2.1 algoritma dasar RRT [12]

Pada **Gambar 2.1** proses algoritma RRT akan berulang sebanyak 1 sampai N iterasi. Algoritma RRT* dapat dilihat pada **Gambar 2.2**. Proses algoritma RRT*

hampir sama dengan algortima RRT perbedaannya terdapat pada *ChooseParent* dan *Rewire*.

Algorithm 2 : $T = (V, E) \leftarrow RRT^*(q_{init})$

1. $T \leftarrow InitializeTree()$
2. $T \leftarrow InsertNode(\emptyset, q_{init}, T)$
3. **for** $k \leftarrow 1$ **to** N **do**
4. $\quad q_{rand} \leftarrow RandomSample(k)$
5. $\quad q_{nearest} \leftarrow NearestNeighbor(q_{rand}, Q_{near}, T)$
6. $\quad q_{new} \leftarrow Steer\ (q_{nearest}, q_{rand}, \Delta q)$
7. $\quad$ **if** $Obstaclefree(q_{new}, q_{nearest})$ **then**
8. $\quad\quad Q_{near} \leftarrow Near(T, q_{new})$
9. $\quad\quad q_{parent} \leftarrow ChooseParent(q_{new}, Q_{near}, q_{nearest})$
10. $\quad\quad T \leftarrow InsertNode(q_{parent}, q_{new}, T)$
11. $\quad\quad T \leftarrow Rewire(T, Q_{near}, q_{parent}, q_{new})$
12. **end**

Gambar 2.2 Algoritma RRT* [12]

Pada **Gambar 2.3** terdapat proses *ChooseParent.* Pada proses ini akan dicari node-node terdekat dari q_{new}. Node-node terdekat dari q_{new} ini dinamakan q_{near} mode dari q_{near} yang membuat jarak antara q_{new} dan node awal minimal, akan dijadikan node parent dari q_{new}.

Algorithm 3 :
$q_{min} \leftarrow ChooseParent(q_{rand}, Q_{near}, q_{nearest}, \Delta q)$

1. $q_{min} \leftarrow q_{nearest}$
2. $c_{min} \leftarrow Cost(q_{nearest}) + c(q_{rand})$
3. **for** $q_{near} \in Q_{near}$ **do**
4. $\quad q_{path} \leftarrow Steer(q_{near}, q_{rand}, \Delta q)$
5. $\quad$ **if** $ObstacleFree(q_{path})$ **then**
6. $\quad\quad c_{new} \leftarrow Cost(q_{near}) + c(q_{rand})$
7. $\quad\quad$ **if** $c_{min} < c_{new}$ **then**
8. $\quad\quad\quad c_{min} \leftarrow c_{new}$
9. $\quad\quad\quad q_{min} \leftarrow q_{new}$
10. $\quad\quad$ **end**
11. $\quad$ **end**
12. **end**
13. **return** q_{min}

Gambar 2.3 operasi *ChooseParent* pada algoritma RRT* [12]

Proses pada algoritma *rewire* dapat dilihat pada **Gambar 2.4.** Proses pada *rewire* akan memeriksa setiap node q_{near} untuk mengetahui apakah menjangkau q_{near} melalui qnew akan menghasilkan jarak yang lebih dekat dengan q_{start}. Apabila dari hasil pemeriksaan tersebut diperoleh jarak yang lebih pendek dari jarak sebelumnya maka program akan melakukan proses *rewire* dengan mengubah *parent* yang ada pada q_{near} menjadi *parent* yang ada pada q_{new}.

Algorithm 4 : $T \leftarrow Rewire(T, Q_{near}, q_{min}, q_{rand})$

1. **for** $q_{near} \in Q_{near}$ **do**
2. $q_{path} \leftarrow Steer(q_{near}, q_{rand}, \Delta q)$
3. **if** $ObstacleFree(q_{path})$ **and** $Cost(q_{rand}) + c(q_{path}) < Cost(q_{near})$ **then**
4. $T \leftarrow ReConnect(q_{rand}, q_{near}, T)$
5. **end**
6. **return** T

Gambar 2.4 operasi *rewire* pada algoritma RRT* [12]

Algoritma Informed-RRT* merupakan pengembangan dari metode *sampling* RRT* yang mana perbedaan tersebut terdapat pada pembatasan area pencarian pada suatu daerah yang berbentuk elips yang melingkupi titik awal dan titik akhir. Algoritma *Informed*-RRT* dapat dilihat pada **Gambar 2.5.** Pembatasan wilayah pencarian tersebut terjadi setelah jalur yang menghubungkan titik awal dengan titik akhir telah ditemukan. Kegunaan dari pembatasan wilayah pencarian tersebut agar memaksimalkan iterasi yang tersisa pada algoritma *Informed*-RRT* untuk terus mencari jalur yang paling optimal. Apabila jarak antara titik awal dan

titik akhir semakin pendek maka luas area yang melingkupi titik awal dan titik akhir akan semakin sempit.

Algorithm 9 Informed RRT*-Connect Algorithm

$V_a \leftarrow \{x_{start}\}$; $E_a \leftarrow \emptyset$;
$V_b \leftarrow \{x_{goal}\}$; $E_b \leftarrow \emptyset$;
$G_a \leftarrow (V_a, E_a)$; $G_b \leftarrow (V_b, E_b)$;
$X_{soln} \leftarrow \emptyset$;
$c_{best} \leftarrow \infty$;
for $i = 1$ *to* n **do**
    $previous\_c_{best} \leftarrow c_{best}$;
    $c_{best} \leftarrow CalculateShortestPathLengh(X_{soln})$;
    if $c_{best} < previous\_c_{best}$ **then**
        $PruneTree(V, E, c_{best})$;
    end if
    $x_{rand} \leftarrow InformedSample(x_{start}, x_{goal}, c_{best})$;
    if $Extend^*(G_a, x_{rand}) \neq Trapped$ **then**
        $Connect^*(G_b, x_{new})$;
    end if
    $Swap(G_a, G_b)$;
    if $isSolutionFound(x_{new})$ **then**
        $X_{soln} \leftarrow X_{soln} \bigcup \{x_{new}\}$
    end if
end for
return G_a, G_b;

Gambar 2.5 Algoritma Informed-RRT*[9]

pada **Gambar 2.5** apabila setiap iterasi ditemukan nilai *Cbest* yang lebih kecil dari sebelumnya maka nilai *Cbest* yang lama akan digantikan dengan nilai *Cbest* yang baru. Algoritma *Extend** dapat dilihat pada **Gambar 2.6**

Algorithm 6 Extend* Function

```
1: function Extend*(G = (V, E), x)
2:     x_nearest ← Nearest(G, x);
3:     x_new ← Steer(x_nearest, x);
4:     if isCollisionFree(x_nearest, x_new) then
5:         V ← V ⋃ {x_new};
6:         x_min ← x_nearest;
7:         X_near ← Near(G, x_new, r_RRT*);
8:         c_min ← Cost(x_nearest, G)
           +Cost(Line(x_nearest, x_new));
9:         for each x_near ∈ X_near \ x_nearest do
10:            if isCollisionFree(x_near, x_new) &
                 (Cost(x_near, G) + Cost(Line(x_near, x_new))
                   < c_min) then
11:                x_min ← x_near;
12:                c_min ← Cost(x_near, G)
                   +Cost(Line(x_near, x_new));
13:            end if
14:        end for
15:        E ← E ⋃ {x_min, x_new};
16:        for each x_near ∈ X_near \ x_min do
17:            if isCollisionFree(x_near, x_new) &
                   (Cost(x_new, G)+Cost(Line(x_new, x_near))
                   < Cost(x_near, G)) then
18:                x_parent ← Parent(x_near, G);
19:                E ← E \ {(x_parent, x_near)};
20:                E ← E ⋃ {(x_new, x_near)};
21:            end if
22:        end for
23:        if (x_new = x) then
24:            return Reached;
25:        else
26:            return Advanced;
27:        end if
28:    end if
29:    return Trapped;
30: end function
```

Gambar 2.6 Algoritma *Extend** [9]

Pada **Gambar 2.6** algoritma *Extend** berfungsi untuk melakukan pencarian node-node terdekat dari *random node* kemudian memeriksa tabrakan pada *obstacle*. Pada *Extend** juga terdapat proses perhitungan untuk mencari jarak terpendek jika dihubungkan dari *random node* menuju titik-titik yang ada pada daftar pohon pecarian. Proses tersebut untuk menentukan *parent* dari node baru yang akan ditambahkan pada daftar node-node pohon pencarian.

2.2 Boundary Sampling

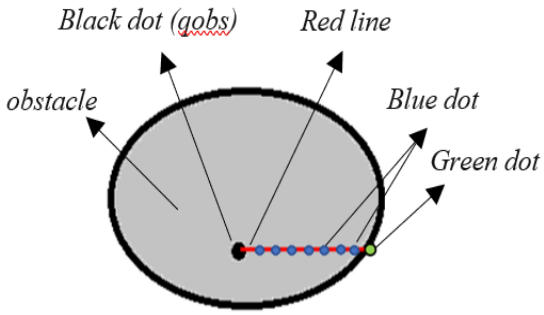
Algoritma *boundary nodes* dapat dilihat pada **Gamber 2.7.**

Algorithm 7: *BoundaryNodes* (*g=0, c, x=false,* arr [ ], n)

```
1  rand q_obs ← random node in obs
2  c= q_obs
3  while g ≤ 360 then
4      while x = false then
5          if c CollisonFree
6              arr [ ] =, c ← add c to array
7              x= true
8          else
9              c+=0,01
10     g+=n
11     x=false
12     c= q_obs
13 return arr [ ]
```

Gambar 2.7 Algoritma *boundary nodes*

pada **Gambar 2.7** proses pada baris ke-1 yaitu membentuk titik acak di dalam *obstacle* (q_{obs}). Proses pada baris ke-2 akan memeriksa apakah sudut g pada titik q_{obs} lebih kecil sama dengan 360 derajat. Proses pada baris ke-4 akan memeriksa nilai x apakah bernilai *true* atau *false*. Proses pada baris ke-5 akan memeriksa apakah c berada berada diluar *obstacle* atau tidak. Apabila nilai c berada di luar *obstacle* maka tambahkan nilai c ke dalam *array* (arr) dan *x* akan menjadi *true*. Jika c masih berada di dalam *obstacle* maka nilai c akan ditambah dengan nilai 0,01 menjauhi titik q_{obs} pada sudut g. Penggunaan nilai 0,01 bertujuan agar pencarian batas *obstacle* menjadi lebih akurat. Semakin kecil nilainya maka akan semakin akurat titik batas *obstacle* yang akan didapatkan. Nilai n merupakan penentu banyaknya titik sampel yang akan diambil pada setiap permukaan *obstacle*. Ilustrasi algoritma pada **Gambar 2.7** dapat dilihat pada **Gambar 2.8.**



Gambar 2.8 ilustrasi pencarian titik batas pada *obstacle*

Titik warna biru pada **Gambar 2.8** merupakan titik-titik pengujian untuk mencari batas dari *obstacle*. Titik biru ini merupakan variabel c yang terdapat pada **Gambar 2.7**. Garis berwarna merah pada **Gambar 2.8** merupakan garis untuk sudut 0 derajat dari titik q_{obs} (*Black dot*). Titik warna hijau pada **Gambar 2.8** merupakan titik sampel batas *obstacle* yang didapatkan setelah beberapa pengujian pada titik biru dilakukan. Algoritma pemilihan titik sampel yang telah di simpan pada variabel *arr []* yang terdapat pada **Gambar 2.7** dapat dilihat pada **Gambar 2.9**.

Algoritma 5: *BoundarySample(arr [ ])*

```
1  rand ← random number; between 0 and 100
2  if rand ≤ 10 then
3      sample ← a RandomConfig();
4  else
5      randarr ← random number; between 0 and
       arr length
6      sample ← arr [randarr]
7  return sample
```

Gambar 2.9 Algoritma *BoundarySample*

Proses pada **Gambar 2.9** baris ke-1 yaitu membangkitkan angka acak (*rand*) dari 0 sampai 100. Kemudian proses pada baris ke-2 yaitu menguji apabila *rand* < 10 maka ambil sampel acak pada ruang pencarian. Proses pada baris ke-4 Apabila *rand > 10 maka* pilih salah satu titik sampel yang terdapat pada variabel *arr* [ ].

2.3 *Goal biasing Sampling*

goal biasing sampling bekerja dengan cara mengambil titik sampel pada ruang pencarian secara acak kemudian sesekali akan mencoba mengambil titik sampel pada titik tujuan. Besarnya pengambilan sampel pada titik tujuan tergantung dari besarnya persentase yang ditetapkan. Algoritma goal biasing sampling dapat dilihat pada **Gambar 2.10**

Algorithm 3 : GoalBiasSAMPLE(q_{goal})

```
1: rand ← RandomNumber            ▷ between 0 and 100
2: if rand < k then
3:     q_rand ← q_goal
4: else
5:     q_rand ← RandomConfig();
6: end if
7: return q_rand
```

Gambar 2.10 Algoritma *goal biasing sampling* [15]

Pada **Gambar 2.10** parameter k merupakan besaran persentase peluang dari algoritma *goal biasing sampling* yang akan mengambil langsung titik sampel pada titik tujuan. Besaran persentase yang direkomendasikan yaitu antara 0% sampai 10% [6], [15].

2.4 Hybrid sampling

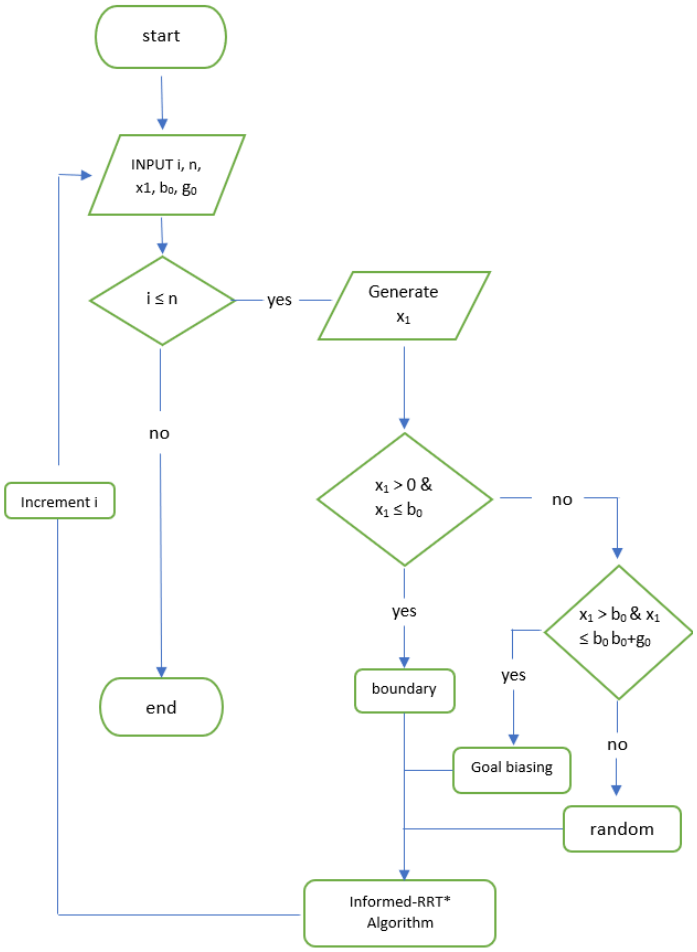
Algoritma *hybrid sampling* merupakan itegrasi dari beberapa metode *sampling*. Pada **Gambar 2.11** dibawah adalah algoritma metode *sampling hybrid* yang merupakan integrasi dari metode *sampling goal biasing*, *boundary sampling*, dan *random sampling.* Algoritma *hybrid sampling* dapat dilihat pada **Gambar 2.11** berikut ini.

Algorithm 1: Hybrid Sampling (b_0, g_0)

```
1  x1 ← a random number;   between 0 and 100
2  if x1 > 0 & x1 <= b0 then
3      sample ← boundary
4  else if x1 > b0 & x1 <= b0 + g0 then
5      sample ← goal biasing
6  else
7      sample ← random
8  end if
9  return sample
```

Gambar 2.11 Algoritma *hybrid sampling*

Pada **Gambar 2.11** metode *sampling hybrid* bekerja dengan cara membangkitkan nilai *random* antara 0-100 bila nilai *random* berada diantara 0 dan b_0 maka *boudary sampling akan aktif.* Apabila nilai *random* berada diantara b_0 dan $b_0 + g_0$ maka *goal biasing sampling* akan aktif. Selain dari kedua kondisi diatas maka *random sampling* yang akan bekerja. *Flowchart* dari metode *hybrid sampling* dapat dilihat pada **Gambar 2.12** dibawah ini



Gambar 2. 12 *Flowchart hybrid sampling*

2.5 Random sampling

Algoritma *random sampling* dapat dilihat pada **Gambar 2.13** dibawah ini

Algorithm 2 : UniformSAMPLE()

1: $q_{rand} \leftarrow$ RandomConfig();
2: **return** q_{rand}

Gambar 2.13 Algoritma *random sampling* [15]

Pada **Gambar 2.13** algoritma *random sampling* berfungsi untuk mengambil sampel secara acak pada ruang pencarian.