BAB II

STUDI LITERATUR

2.1 Rapidly Exploring Random Tree

Rapidly Exploring Random Tree dikembangkan untuk memecahkan kendala nonholomy dan dinamis, algoritma *Rapidly Exploring Random Tree* adalah algoritma berdasarkan struktur pohon. Algoritma ini sangat cocok untuk menyelesaikan masalah kinematika dan perencanaan jalur, karena kurva yang dapat dieksekusi disimulasikan melalui kinematika dan pasangan *node* [15]. *Pseudo code* algoritma *Rapidly Exploring random Tree* dapat dilihat pada **Gambar 2.1** dibawah,

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 *Pseudo code* Algoritma *Rapidly Exploring random Tree* [16]

Pada **Gambar 2.1** terdapat beberapa proses pada algoritma *Rapidly Exploring Random Tree*, yaitu *RandomSample* yang berfungsi mengambil sampel pada ruang kerja, disebut sebagai q*rand. NearestNeightbor* berfungsi untuk mencari *node* terdekat ke *qrand* pada pohon pencarian, disebut sebagai *qnearest.* Selanjutnya adalah *qnew* yang berfungsi untuk membangun *node* yang baru diantara *qrand* dan *qnearest* yang berjarak q dari *qnearest. qnew*

ditambahkan pada pohon pencarian apabila tidak ada hambatan diantara *qnearest* dan *qnew* [16].

2.2 *Rapidly Exploring Random Tree**

*Rapidly Exploring Random Tree** algoritma berbasis *sampling* tambahan yang menemukan jalur awal dengan sangat cepat, kemudian mengoptimalkan jalur saat eksekusi berlangsung [2]. **Gambar 2.2** di bawah adalah *Pseudo code* algoritma *Rapidly Exploring Random Tree**.

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

```
1  T← InitializeTree();
2  T← InsertNode(Ø, z_init, T);
3  for i=0 to i=N do
4      z_rand ← Sample(i);
5      z_nearest ← Nearest(T, z_rand);
6      (x_new, u_new, T_new) ← Steer (z_nearest, z_rand);
7    if Obstaclefree(x_new) then
8      Z_near ← Near(T, z_new, |V|);
9      z_min ← Chooseparent (Z_near, z_nearest, z_new, x_new);
10       T← InsertNode(z_min, z_new, T);
11       T← Rewire (T, Z_near, z_min, z_new);
12 return T
```

Gambar 2.2 *Pseudo code* Algoritma *Rapidly Exploring Random Tree** [17]

Gambar 2.2 di atas adalah *pseudo code* dari algoritma *Rapidly Exploring Random Tree**, yang membedakan dari algoritma *Rapidly Exploring Random Tree* adalah adanya sub program *Chooseparent* dan *Rewire*.

2.3 *Goal Biassing Sampling*

Metode *sampling Goal Biassing* adalah algoritma berbasis pengambilan sampel yang mampu mengidentifikasi solusi awal dalam waktu yang lebih singkat daripada algoritma *Rapidly Exploring Random Tree,* serta

menunjukkan peningkatan yang signifikan dalam efisiensi komputasi. **Gambar 2.3** di bawah adalah *Pseudo code* untuk metode *sampling Goal Biassing* [18].

Algorithm 3 : GoalBiasSAMPLE(q_{goal})

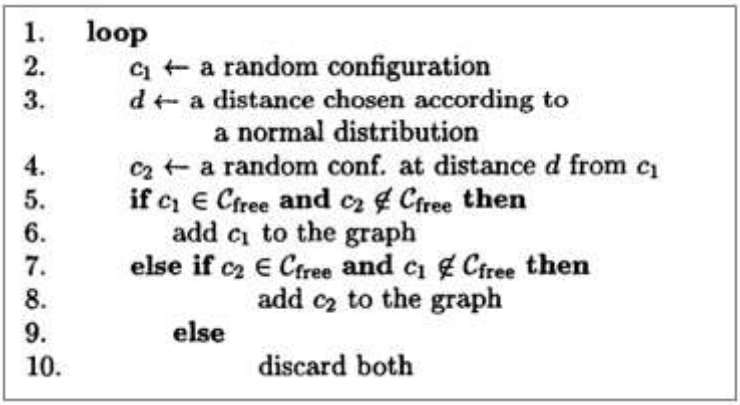
1: $rand \leftarrow$ RandomNumber ▷ between 0 and 100
2: **if** $rand < k$ **then**
3: $q_{rand} \leftarrow q_{goal}$
4: **else**
5: $q_{rand} \leftarrow$ RandomConfig();
6: **end if**
7: **return** q_{rand}

Gambar 2.3 Algoritma metode *sampling Goal Biassing* [18]

Pada **Gambar 2.3** di atas, algoritma mendefinisikan konfigurasi tujuan (*qgoal*) dan bilangan acak dalam algoritma yang dipilih antara angka 0 hingga 100. Jika bilangan acak (*rand)* kurang dari *k* maka konfigurasi acak *qrand* akan menjadi konfigurasi tujuan *qgoal.* Jika bilangan acak lebih dari *k* maka konfigurasi acak *qrand* akan memilih metode *sampling* seragam (*uniform sampling).* Persentase bias yang direkomendasikan adalah berkisar antara 0% hingga 10% [18].

2.4 *Gaussian Sampling*

Pengambilan sampel *Gaussian* dimaksudkan untuk menambahkan lebih banyak sampel di dekat *obstacle,* mengambil dua sampel acak, dimana jarak antara sampel dipilih menurut distribusi *Gaussian*. Jika salah satu sampel terletak di dalam *obstacle,* kemudian sampel yang lain berada di luar *obstacle,* maka diambil titik di antara keduanya untuk dijadikan titik baru [19]. Algoritma dari *Gaussian sampling* dapat dilihat pada **Gambar 2.4** di bawah

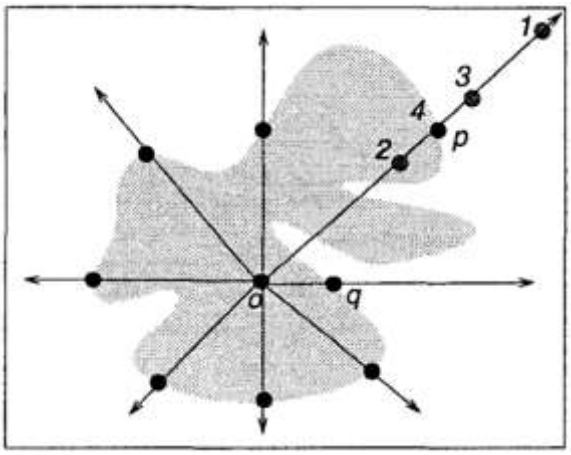


Gambar 2.4 Algoritma *Gaussian sampling* (diadaptasi dari [19])

Pada **Gambar 2.4**, *Gaussian sampling* akan ditambahkan ke dalam grafik hanya jika menemukan konfigurasi terlarang di dekatnya, yaitu jika salah satunya berada di dalam *obstacle*, dan yang lainnya berada di luar obstacle. Jika dibandingkan dengan *random sampling*, *Gaussian sampling* lebih lambat karena menguji dua konfigurasi untuk menambahkan ke grafik, dan tidak ditambahkan ketika keduanya berada di luar *obstacle* maupun di dalam *obstacle* [19].

2.5 *Boundary Sampling*

Boundary sampling mengambil *node-node* pada daerah batasan *obstacle*, dengan cara menentukan terlebih dahulu satu *node* di dalam *obstacle*, kemudian *node* tersebut akan terdistribusi secara merata sebanyak *m. Node* yang telah mencapai bagian luar *obstacle* akan dijadikan sebagai *node* penghubung [20]. **Gambar 2.5** memperlihatkan distribusi *node-node m* pada permukaan *obstacle.*



Gambar 2.5 Distribusi *node m* pada permukaan *obstacle* [20]

Pada **Gambar 2.5** pencarian *node* dilakukan dengan mempersempit daerah pada *obstacle* yang diketahui mengandung titik permukaan, titik awal o adalah titik yang berada di dalam *obstacle*, dan titik berlabel 1 adalah titik terluar dari *obstacle.* Selanjutnya, pencarian dilakukan hingga sebuah titik pada permukaan *obstacle* ditemukan [20].