Penerapan dan Perbedaan UCS dan A* dalam Pembuatan AI untuk Bot dalam Gim Tiga Dimensi

Akbar Al Fattah - 13522036
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13522036@std.stei.itb.ac.id

***Abstrak*—Bot adalah salah satu mekanik game yang sering digunakan untuk membuat video game semakin seru. Namun, dibalik sebuah bot, terdapat sebuah AI untuk mencari jalur /*pathfinding*. Ada banyak sekali algoritma *pathfinding*, seperti BFS, DFS, A*, UCS, GBFS, Dijkstra, dan masih banyak lagi. Makalah ini akan membahas tentang penerapan algoritma A* dan UCS dalam pembuatan AI bot dalam gim tiga dimensi dan membandingkan mana algoritma yang lebih cocok secara performa di dalam gim tiga dimensi dan *path* yang dipilih.**

Kata Kunci bot, A*, UCS, gim, AI, graf, grid, kubus, diagonal, sumbu, performa.

I. PERKENALAN

Industri video game adalah salah satu dari media hiburan yang sangat berkembang pesat di era zaman sekarang. Video game dapat dinikmati di berbagai platform, seperti konsol, komputer personal (PC), *platform mobile*, seperti ponsel pintar, tablet, bahkan di perangkat *virtual reality* (VR).

Saat awal perkembangannya, video game hanya mampu menampilkan visual dalam bentuk 2 dimensi. Seiring perkembangan zaman, video game berkembang sehingga mampu memiliki grafis 3 dimensi.

Setiap video game memiliki mekanik-mekanik yang merupakan bagian dari desain game yang dibuat khusus untuk game tersebut. Hal ini bertujuan agar pemain game tersebut merasakan keseruan dan menyukai game tersebut.

Salah satu bagian dari mekanik game adalah *bot*. *Bot* adalah objek di dalam game yang berperan mirip seperti pemain, namun tidak dikendalikan secara manual oleh pemain dan mampu bergerak sendiri secara otomatis. Oleh karena *bot* ini tidak digerakkan oleh pemain secara langsung, maka *bot* ini memerlukan AI / kecerdasan buatan agar mampu untuk bergerak sendiri secara otomatis.

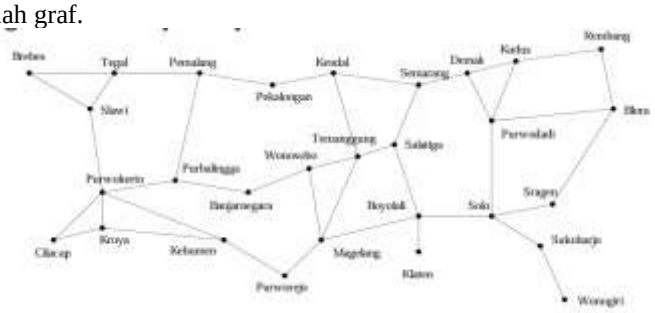
Untuk AI yang dibutuhkan oleh bot tersebut, diperlukan sebuah algoritma *pathfinding* agar bot bisa bergerak sendiri. Algortima *pathfinding* yang umum digunakan adalah A*, UCS, Dijkstra, BFS, DFS, dan GBFS. Namun, di makalah ini, yang akan di bahas adalah algoritma A* dan UCS (ekuivalen jika dengan algoritma Dijkstra).

Pada makalah ini, demonstrasi algoritma dilakukan menggunakan kakas Unity dan semua source code ditulis dalam bahasa C#.

II. DASAR TEORI

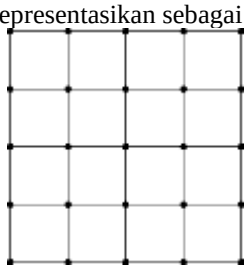
A. Teori Graf

Graf adalah sebuah struktur data yang merepresentasikan hubungan antara objek-objek diskrit. Berikut adalah contoh dari sebuah graf.

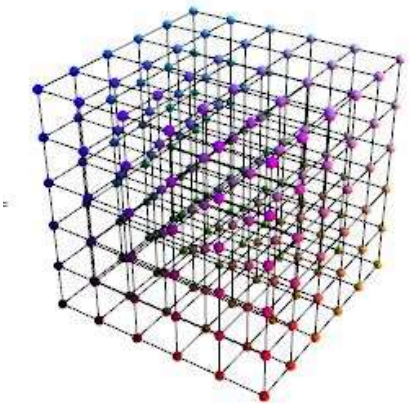


Gambar 1: Contoh Gambar Graf
Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/19-Graf-Bagian1-2023.pdf

Selain dalam bentuk graf di atas, graf juga dapat direpresentasikan sebagai petak/*grid* 2 dimensi atau 3 dimensi.



Gambar 2: Contoh Gambar Grid 2 dimensi
Sumber: https://www.researchgate.net/figure/The-two-dimensional-grid_fig1_328757809



Gambar 3: Contoh Gambar Grid 3 dimensi
Sumber:
https://stackoverflow.com/questions/33577310/how-to-best-represent-a-grid-graph-in-3d-euclidean-space

*B. Algoritma A**

Algoritma A* adalah salah satu algoritma pencarian *informed search* karena algoritma A* menggunakan nilai heuristik yang *admissible* sebagai bahan pertimbangan untuk pencariannya. Nilai-nilai yang digunakan dalam algoritma A* adalah:

1. g(n), cost total untuk mencapai node n dari node awal.
2. h(n), cost total untuk mencapai node tujuan dari node n
3. f(n), f(n) = g(n) + h(n)

Agar algoritma A* dapat berfungsi sebagaimana mestinya, maka haruslah $h(n) \leq {}^{*}h(n)$ dengan ${}^{*}h(n)$ adalah cost terendah sesungguhnya dari n ke node tujuan.

Berikut ini adalah algoritma A* secara umum.

1. **Inisialisasi Node:**
   - Tentukan node awal dan node tujuan berdasarkan posisi sebenarnya di ruang 3 dimensi Unity
2. **Inisialisasi List Node yang masih dianggap Open dan Closed :**
   - Buat daftar kosong untuk *list open node* (node yang belum dikunjungi) dan *list closed node* (node yang sudah dikunjungi).
   - Tambahkan node awal ke *list open node*.
3. **Mulai Algoritma A***
   - Selama ada node dalam list *open node*, lanjutkan proses pencarian.
4. **Pilih Node dengan *cost* Terendah:**
   - Dari node-node dalam open list, pilih node dengan f(n) terkecil (fCost). Jika ada node dengan nilai f(n) yang sama, pilih node dengan nilai heuristik (h(n)) yang lebih kecil.
5. **Pindahkan Node dari Open ke Closed List:**
   - Hapus node yang dipilih dari open list dan tambahkan ke closed list.
6. **Periksa Apakah Node Sekarang adalah Node Tujuan:**
   - Jika node yang dipilih adalah node tujuan, artinya kita sudah mendapatkan pathnya
   - Hentikan pencarian
7. **Periksa Semua Tetangga dari Node Sekarang:**
   - Untuk setiap node tetangga dari node sekarang:
     - Lewati node tetangga yang tidak bisa dilalui atau yang sudah ada dalam closed list.
8. **Hitung cost ke Node Tetangga:**
   - Hitung g(n) dari node sekarang ke nodetetangga.
   - Jika biaya perjalanan ke tetangga lebih kecil dari biaya yang sudah ada atau node tetangga belum ada di dalam open list:
     - Perbarui g(n) dan h(n) dari node tetangga.
     - Tetapkan node sekarang sebagai parent dari node tetangga.
     - Jika tetangga belum ada di dalam open list, tambahkan ke open list.
9. **Ulangi Proses:**
   - Ulangi langkah-langkah ini sampai menemukan jalur ke node tujuan atau open list kosong. Jika open list kosong dan node tujuan belum ditemukan, berarti tidak ada *path* terpendek.

C. Algoritma UCS

Algoritma UCS adalah kasus khusus dari algoritma A* yang tidak menganggap nilai heuristik (nilai fungsi h(n)) ada (atau semua nilai h(n) dianggap sama dengan 0).

Catatan: Pada kasus ini, algortima UCS secara algoritma ekuivalen dengan algoritma Dijkstra karena kedua algoritma ini pada dasarnya memiliki logika yang sama, namun perbedaannya hanya terletak di cara menyimpan node selanjutnya.

Berikut ini adalah algoritma UCS secara umum.

1. **Inisialisasi Node:**
   - Tentukan node awal dan node tujuan berdasarkan posisi sebenarnya di ruang 3 dimensi Unity
2. **Inisialisasi List Node yang masih dianggap Open dan Closed :**
   - Buat daftar kosong untuk *list open node* (node yang belum dikunjungi) dan *list closed node* (node yang sudah dikunjungi).
   - Tambahkan node awal ke *list open node*.
3. **Mulai Algoritma UCS**
   - Selama ada node dalam list *open node*, lanjutkan proses pencarian.
4. **Pilih Node dengan *cost* Terendah:**
   - Dari node-node dalam open list, pilih node dengan g(n) terkecil.
5. **Pindahkan Node dari Open ke Closed List:**
   - Hapus node yang dipilih dari open list dan tambahkan ke closed list.
6. **Periksa Apakah Node Sekarang adalah Node Tujuan:**
   - Jika node yang dipilih adalah node tujuan, artinya kita sudah mendapatkan pathnya
   - Hentikan pencarian
7. **Periksa Semua Tetangga dari Node Sekarang:**
   - Untuk setiap node tetangga dari node sekarang:
     - Lewati node tetangga yang tidak bisa dilalui atau yang sudah ada dalam closed list.
8. **Hitung cost ke Node Tetangga:**

- Hitung g(n) dari node sekarang ke nodetetangga.
- Jika biaya perjalanan ke tetangga lebih kecil dari biaya yang sudah ada atau node tetangga belum ada di dalam open list:
    - Perbarui g(n) dari node tetangga.
    - Tetapkan node sekarang sebagai parent dari node tetangga.
    - Jika tetangga belum ada di dalam open list, tambahkan ke open list.

9. **Ulangi Proses:**
    - Ulangi langkah-langkah ini sampai menemukan jalur ke node tujuan atau open list kosong. Jika open list kosong dan node tujuan belum ditemukan, berarti tidak ada *path* terpendek.

III. IMPLEMENTASI

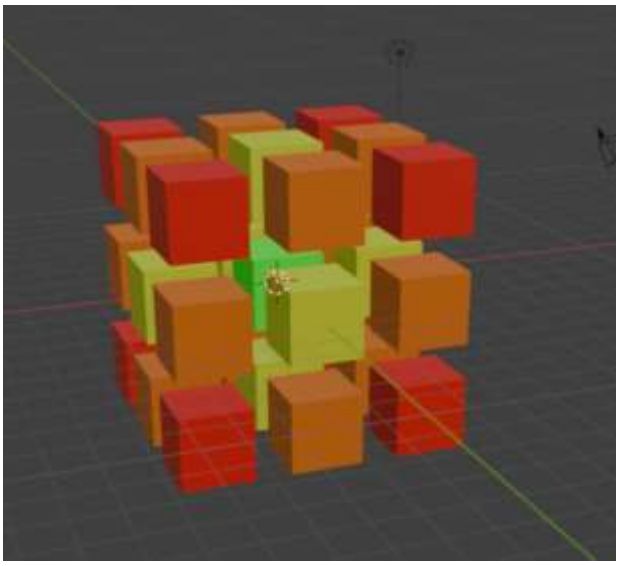
A. Representasi graf

Pada makalah ini, graf digambarkan sebagai sebuah grid 3 dimensi. Di dalam grid tersebut, terdapat kubus-kubus yang dapat dianggap sebagai satu buah *node*/simpul.

Pada grid 3 dimensi, ada 3 macam cara untuk melakukan perpindahan antarsimpul, yaitu:

1. Pergerakan searah dengan sumbu X,Y, dan Z
2. Pergerakan menuju diagonal bidang dari suatu subkubus
3. Pergerakan menuju diagonal ruang dari suatu subkubus.

Untuk memperjelas ilustrasi, perhatikan gambar ilustrasi suatu subgrid/subkubus yang sudah diberi kode warna di *Blender* berikut.

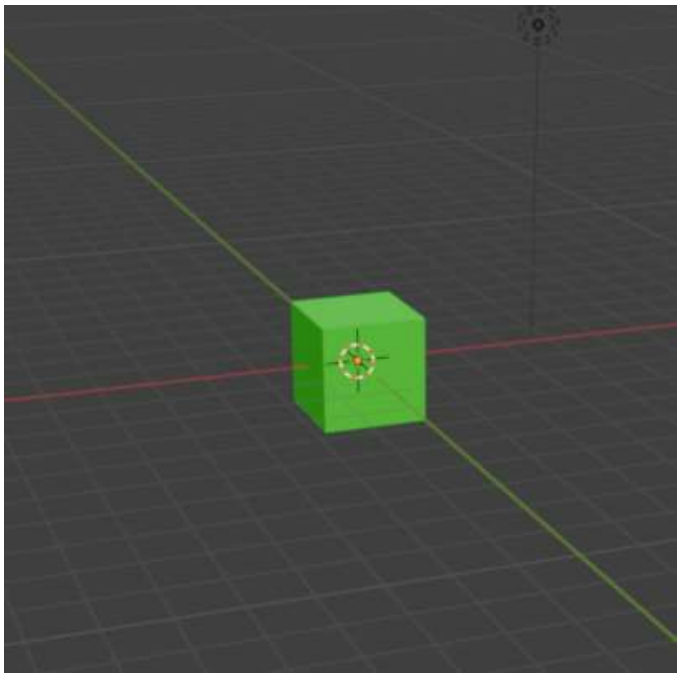


Gambar 4: Ilustrasi grid 3 dimensi dengan kode warna di Blender

Sumber: Dokumentasi penulis

Agar lebih jelas, berikut adalah *breakdown* dari warna setiap kubus.

1. Kubus Hijau

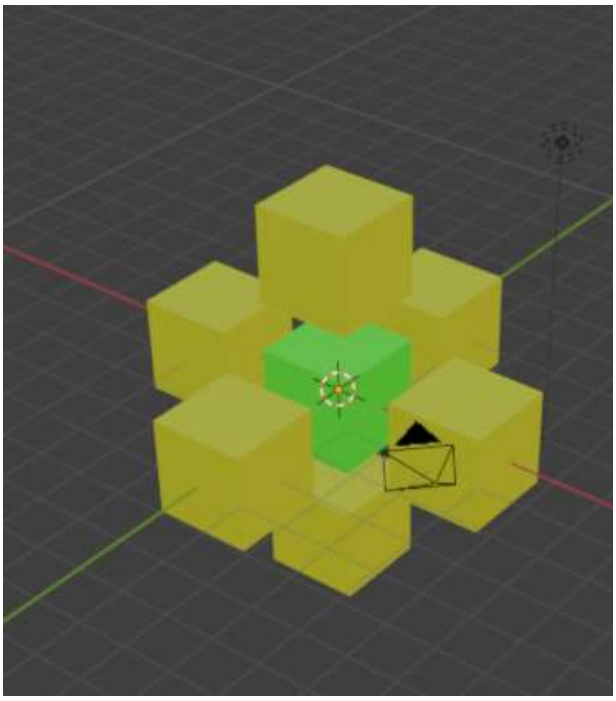


Gambar 5: Ilustrasi kubus hijau di Blender

Sumber: Dokumentasi penulis

Kubus hijau pada subgrid/subkubus ini merepresentasikan posisi objek/bot/pemain saat ini. Untuk gambar kubus warna lain pada laporan ini, kubus hijau ini akan tetap ditampilkan sebagai acuan posisi bot saat ini.

2. Kubus Kuning

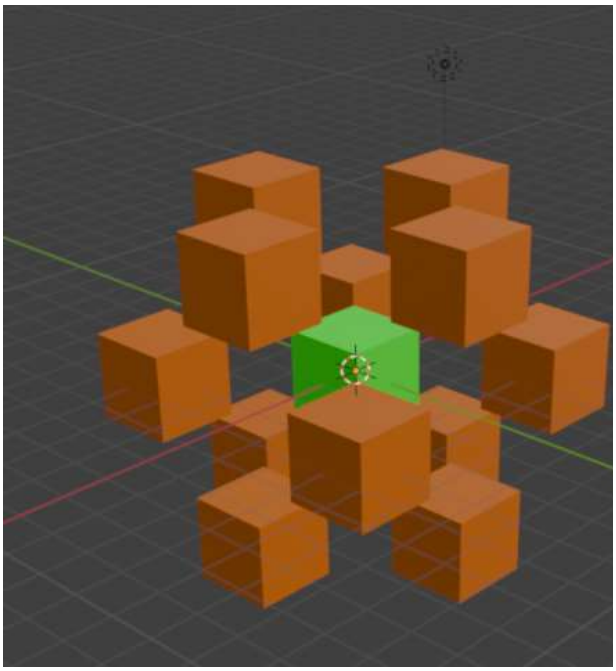


Gambar 6: Ilustrasi kubus kuning di Blender

Sumber: Dokumentasi penulis

Kubus kuning pada subgrid/subkubus ini merepresentasikan semua *node*/simpul yang dapat dikunjungi dengan melakukan pergerakan searah dengan sumbu X, sumbu Y, dan sumbu Z.

3. Kubus Jingga

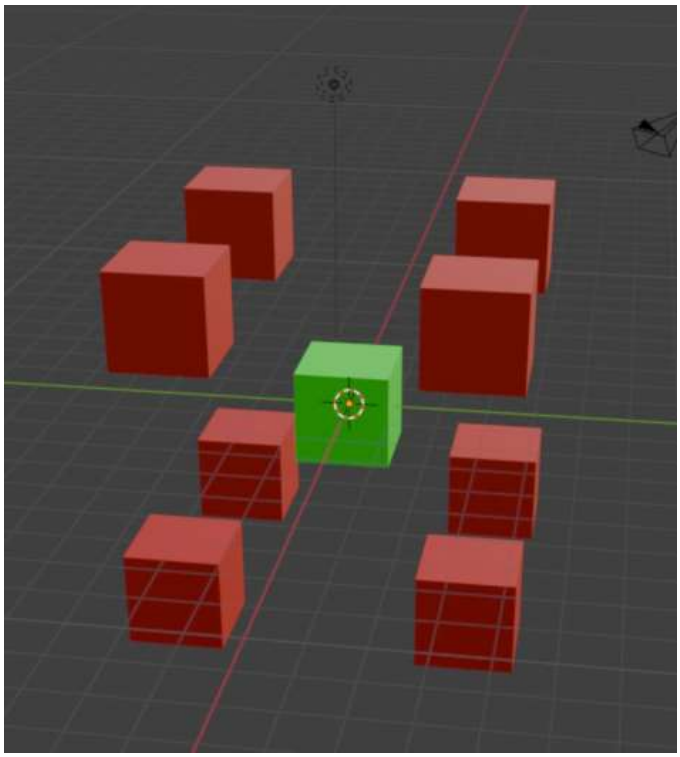


Gambar 7: Ilustrasi kubus jingga di Blender

Sumber: Dokumentasi penulis

Kubus jingga pada subgrid/subkubus ini merepresentasikan semua *node*/simpul yang dapat dikunjungi dengan melakukan pergerakan menuju semua diagonal bidang dari suatu subgrid/subkubus.

4. Kubus Merah

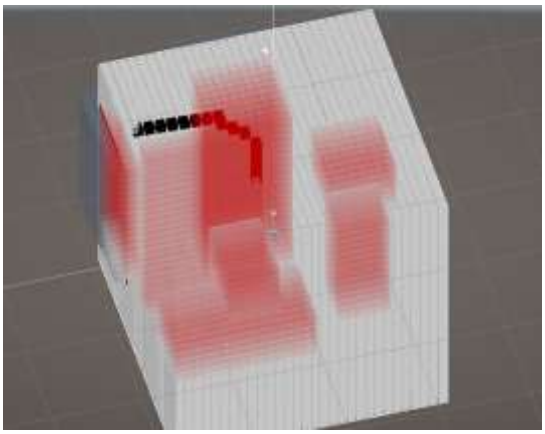


Gambar 8: Ilustrasi kubus merah di Blender

Sumber: Dokumentasi penulis

Kubus merah pada subgrid/subkubus ini merepresentasikan semua *node*/simpul yang dapat dikunjungi dengan melakukan pergerakan menuju semua diagonal ruang dari suatu subgrid/subkubus.

Pada Unity, terdapat fitur *Layer* yang bertujuan untuk membedakan mana objek yang tidak bisa dilewati oleh bot (penghalang/*obstacle*). Untuk membedakan mana kubus yang merepresentasikan obstacle dan mana yang bukan, Unity menyediakan fitur gizmo untuk melakukan *preview obstacle*, perhatikan gambar berikut.



Gambar 9: Gizmo representasi grid 3 dimensi di Unity

Sumber: Dokumentasi penulis

Gambar di atas terdiri atas banyak kubus yang dibuat oleh gizmo di Unity.

Berikut keterangan warna gizmo pada gambar di atas.

1. Gizmo kubus merah: Obstacle/penghalang

   Kubus merah tidak dapat dilewati oleh bot karena dianggap sebagai penghalang.

2. Gizmo kubus putih: node yang kosong dan tidak ada obstacle.

   Kubus putih dapat dilewati oleh bot karena tidak ada penghalang.

3. Gizmo kubus hitam: Path yang dipilih oleh bot

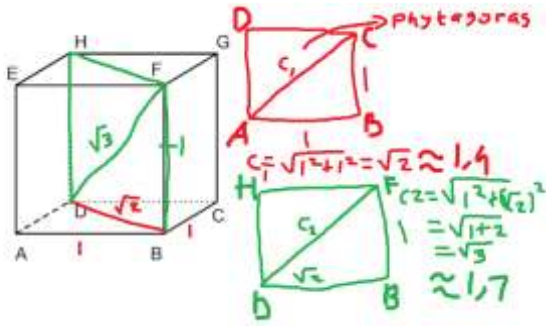
   Kubus hitam adalah path yang dipilih oleh bot. Karena bot menggunakan algoritma A* atau UCS, kubus hitam adalah path yang dianggap path terpendek dari titik asal bot ke tempat tujuan.

B. Pembobotan jarak antar node/simpul/kubus

Seperti yang sudah disebutkan pada bagian III.A., ada 3 cara untuk melakukan pergerakan di dalam grid 3 dimensi, yaitu:

1. Pergerakan searah dengan sumbu X,Y,dan Z. (semua kubus kuning)
2. Pergerakan menuju diagonal bidang dari suatu subkubus. (semua kubus jingga)
3. Pergerakan menuju diagonal ruang dari suatu subkubus. (semua kubus jingga)

Untuk menentukan bobot tiap pergerakan di atas, perhatikan gambar di bawah ini.



Gambar 10: Penentuan bobot jarak antarkubus
Sumber: Dokumentasi penulis

Anggap jarak terpendek ke kubus kuning nilainya 1, yaitu panjang sisi dari kubus satuan itu sendiri. Maka, pada gambar di atas, untuk mencari jarak terpendek dari kubus hijau ke kubus jingga, kita harus mencari panjang dari diagonal bidang dari kubus di atas.

Dari gambar di atas, panjang diagonal bidang adalah:

$$Diagonal\ Bidang = BD = AC$$
$$= \sqrt{AB^2 + BC^2}$$
$$= \sqrt{1^2 + 1^2}$$
$$= \sqrt{2} \approx 1{,}4$$

Setelah kita mengetahui diagonal bidang, kita harus mencari jarak terpendek dari kubus hijau ke kubus merah. Untuk mendapatkan bobot dari kubus merah, kita harus mencari panjang diagonal ruang.

Dari gambar di atas, panjang diagonal ruang adalah:

$$Diagonal\ Ruang = DF = BH$$
$$= \sqrt{BD^2 + BF^2}$$
$$= \sqrt{(\sqrt{2}^2) + 1^2}$$
$$= \sqrt{3} \approx 1{,}7$$

Setelah menghitung nilai dari diagonal bidang dan diagonal ruang, dapat disimpulkan bahwa.

1. Jarak terpendek dari kubus hijau ke kubus kuning manapun adalah 1
2. Jarak terpendek dari kubus hijau ke kubus jingga manapun adalah 1,4
3. Jarak terpendek dari kubus hijau ke kubus merah manapun adalah 1,7

Akan tetapi, komputer akan menghasilkan galat/*error* jika kita menggunakan nilai pecahan desimal/*float* sebagai bobot antarkubus, yang akan mengakibatkan perhitungan jarak menjadi kurang akurat. Untuk mengatasi hal tersebut, penggunaan tipe data float lebih baik dihindari dengan cara semua bobot yang telah didefinisikan di atas dikalikan 10 yang akan membuat nilai-nilai bobot di atas menjadi bertipe bilangan bulat, sehingga penggunaan tipe data float dihindari.

Semua bobot di atas setelah dikalikan dengan 10 berubah menjadi:

1. Jarak terpendek dari kubus hijau ke kubus kuning manapun adalah 10
2. Jarak terpendek dari kubus hijau ke kubus jingga manapun adalah 14

3. Jarak terpendek dari kubus hijau ke kubus merah manapun adalah 17

C. *Definisi g(n), h(n) dan* $^*h(n)$

Pada penerapan algoritma A* dan UCS di makalah ini, nilai g(n) didefinisikan sebagai jarak sebenarnya dari node awal sampai node n dengan menganggap bahwa ada *obstacle* di dalam grid tersebut.
Sementara itu, nilai h(n) yang digunakan pada penerapan ini didefinisikan sebagai jarak dari node n ke node tujuan **tanpa menganggap bahwa ada *obstacle* di dalam grid tersebut**.
Penggunaan h(n) ini *admissible* karena h(n) akan selalu lebih kecil atau sama dengan $^*h(n)$.
Alasan bahwa h(n) $\leq$ $^*h(n)$ selalu benar adalah h(n) adalah jarak dari n ke node tujuan **dengan asumsi *obstacle*/penghalang tidak ada**, sedangkan $^*h(n)$ adalah jarak sesungguhnya dari n ke node tujuan **dengan arti bahwa** $^*h(n)$ **menganggap *obstacle*/penghalang ada**, sehingga h(n) akan cenderung memilih untuk memilih kubus pada diagonal bidang/diagonal ruang secara langsung dibandingkan dengan $^*h(n)$ yang harus melewati penghalang dengan cara memutari penghalang terlebih dahulu. Hal ini akan membuat h(n) selalu lebih kecil atau sama dengan $^*h(n)$.

D. *Source Code Node.cs*

Berikut ini adalah source code untuk representasi dari sebuah node pada grid 3 dimensi yang digunakan pada makalah ini.

```
public class Node
{
    public bool walkable; //menentukan apakah node/kubus ini obstacle atau bukan. (Jika node ini adalah obstacle, walkabble = false. Jika bukan, walkable = true)
    public Vector3 worldPosition; //Posisi grid sebenarnya di ruang 3 dimensi di Unity
    public int gCost,hCost; //nilai gCost dan hCost
    public int gridX,gridY,gridZ; //Posisi node satuan di grid pada sumbu X,Y,dan Z
    public int fCost{ //nilai fCost = gCost + hCost
        get{
            return gCost+hCost;
        }
    }

    public Node parent; //node sebelumnya untuk konstruksi path
    public Node(bool walkable,Vector3 worldPos,int gridX, int gridY, int gridZ){ //Konstruktor Node
        this.walkable = walkable;
        worldPosition = worldPos;
        this.gridX = gridX;
        this.gridY = gridY;
        this.gridZ = gridZ;
    }
}
```

Gambar 11: Source code Node.cs

Sumber: Dokumentasi penulis

E. *Source Code GridCustom.cs*

Berikut ini adalah source code untuk representasi dari grid 3 dimensi yang digunakan dan pembentukan grid 3 dimensi pada makalah ini.

```
public class GridCustom : MonoBehaviour
{
    //ini hanyalah potongan kode dari GridCustom.cs yang hanya berisi method untuk inisialisai dan pembentukan grid 3 dimensi, method lain akan ditambahkan secara terpisah di makalah ini.
    public Vector3 gridSize; //ukuran grid
    public float nodeRadius; //jari-jari tiap node (untuk menentukan ukuran tiap node)
    public LayerMask unwalkableLayer; //Layer obstacle yang membuat bot tidak bisa melewati objek ini
    Node[,,] grid; //representasi grid

    float nodeDiameter; //diameter dari node
    int gridSizeX, gridSizeY, gridSizeZ; //gridSize untuk setiap sumbu dalam tipe integer untuk pembuatan array 3 dimensi
    //berfungsi sebagai koordinat untuk setiap kubus pada grid

    void Start() //fungsi Start pada Unity untuk melakukan inisialisasi
    {
        nodeDiameter = nodeRadius * 2;
        gridSizeX = Mathf.RoundToInt(gridSize.x / nodeDiameter);
        gridSizeY = Mathf.RoundToInt(gridSize.y / nodeDiameter);
        gridSizeZ = Mathf.RoundToInt(gridSize.z / nodeDiameter);
        CreateGrid();
```

```
    }
```

Gambar 12: Potongan Source Code GridCustom.cs

Sumber: Dokumentasi penulis

F. Source Code Fungsi Tambahan Pathfinding

Berikut adalah source code untuk fungsi-fungsi pembantu tambahan untuk algoritma *pathfinding*.

```
    void CreateGrid()
    {
        grid = new Node[gridSizeX, gridSizeY,
gridSizeZ]; //buat grid baru dengan ukuran
gridSizeX *gridSizeY * gridSizeZ
        //tentukan posisi pojok kiri bawah untuk
referensi koordinat grid
        Vector3 worldBottomLeft =
transform.position - Vector3.right * gridSize.x /
2 - Vector3.forward * gridSize.z / 2 - Vector3.up
* gridSize.y / 2;
        for (int x = 0; x < gridSizeX; x++)
        {
            for (int y = 0; y < gridSizeY; y++)
            {
                for (int z = 0; z < gridSizeZ;
z++)
                {
                    //tentukan posisi node pada
ruang 3 dimensi di Unity
                    Vector3 worldPoint =
worldBottomLeft + Vector3.right * (x *
nodeDiameter + nodeRadius) + Vector3.up * (y *
nodeDiameter + nodeRadius) + Vector3.forward * (z
* nodeDiameter + nodeRadius);
                    //Periksa apakah node tersebut
adalah obstacle atau bukan dengan mengecek bola
dengan ukuran radius sama dengan radius node
                    bool walkable
= !Physics.CheckSphere(worldPoint, nodeRadius,
unwalkableLayer);
                    //tambahkan node baru ke grid
                    grid[x, y, z] = new
Node(walkable, worldPoint, x, y, z);
                }
            }
        }
    }
}
```

```
    public List<Node> getNeighbors(Node node)
//fungsi untuk mendapatkan semua node neighbor
dari node posisi pemain saat ini di Unity
    {
        List<Node> neighbors = new List<Node>();

        for (int x = -1; x <= 1; x++)
        {
            for (int y = -1; y <= 1; y++)
            {
                for (int z = -1; z <= 1; z++)
                {
                    if (x == 0 && y == 0 && z ==
0)
                    {
                        continue;
                    }
                    int checkX = node.gridX + x;
                    int checkY = node.gridY + y;
                    int checkZ = node.gridZ + z;
                    if (checkX >= 0 && checkX <
gridSizeX && checkY >= 0 && checkY < gridSizeY &&
checkZ >= 0 && checkZ < gridSizeZ)
                    {
                        neighbors.Add(grid[checkX,
checkY, checkZ]);
                    }
                }
            }
        }
        return neighbors;
    }
```

```
    public TextMeshProUGUI text;
    public bool onlyDisplayPath;
    public List<Node> path;
    public Transform player;
    public Node NodeFromWorldPoint(Vector3
worldPosition) //fungsi untuk mendapatkan node
dari posisi sebenarnya di Unity
    {
        float percentX = (worldPosition.x +
gridSize.x / 2) / gridSize.x;
        float percentY = (worldPosition.y +
gridSize.y / 2) / gridSize.y;
```

```
        float percentZ = (worldPosition.z +
gridSize.z / 2) / gridSize.z;
        percentX = Mathf.Clamp01(percentX);
        percentY = Mathf.Clamp01(percentY);
        percentZ = Mathf.Clamp01(percentZ);

        int x = Mathf.RoundToInt((gridSizeX - 1) *
percentX);
        int y = Mathf.RoundToInt((gridSizeY - 1) *
percentY);
        int z = Mathf.RoundToInt((gridSizeZ - 1) *
percentZ);
        return grid[x, y, z];
    }
```

```
    List<Node> retracePath(Node startNode,
Node endNode) //fungsi untuk mendapatkan
path dari awal sampai akhir
    {
        List<Node> path = new List<Node>();
        Node curNode = endNode;

        while (curNode != startNode)
        {
            path.Add(curNode);
            curNode = curNode.parent;
        }

        path.Reverse();
        grid.path = path;
        return path;
    }

    int getDist(Node nodeA, Node
nodeB)//fungsi untuk menghitung jarak antar
node
    {
        //Pembobotan jarak sudah di bahas
pada bagian III.B.
        int distX = Mathf.Abs(nodeA.gridX -
nodeB.gridX);
        int distY = Mathf.Abs(nodeA.gridY -
nodeB.gridY);
        int distZ = Mathf.Abs(nodeA.gridZ -
nodeB.gridZ);
```

```
        int maxDist = Mathf.Max(distX,
distY, distZ);
        int midDist = distX + distY + distZ
- maxDist - Mathf.Min(distX, distY, distZ);
        int minDist = Mathf.Min(distX,
distY, distZ);

        return 17 * minDist + 14 * (midDist
- minDist) + 10 * (maxDist - midDist);
    }
```

Gambar 13: Source code fungsi tambahan untuk pathfinding
Sumber: Dokumentasi Penulis

G. *Source Code Algoritma A**

Berikut adalah source code untuk implementasi algoritma A*.

```
    public Transform seeker, target;
    GridCustom grid;

    void Awake()
    {
        grid = GetComponent<GridCustom>();
    }

    void Update()
    {
        FindPath(seeker.position,
target.position);
    }
```

```
    void FindPath(Vector3 startPos, Vector3
endPos)
    {
        Stopwatch stopwatch = new
Stopwatch();
        stopwatch.Start();
        // A*
        Node startNode =
grid.NodeFromWorldPoint(startPos); //node
awal
        Node endNode =
grid.NodeFromWorldPoint(endPos); //node
tujuan

        List<Node> openNode = new
List<Node>(); //node yang belum dikunjungi
```

```
        HashSet<Node> closedNode = new
HashSet<Node>(); // node yang sudah
dikunjungi
        openNode.Add(startNode);//masukkan
ke open node

        while (openNode.Count > 0)
        {
            Node curNode = openNode[0];
//node sekarang

            for (int i = 1; i <
openNode.Count; i++)
            {
                if (openNode[i].fCost <=
curNode.fCost && openNode[i].hCost <
curNode.gCost)
                { // ganti node sekarang
(A*)
                    curNode = openNode[i];
                }
            }

            openNode.Remove(curNode);
            closedNode.Add(curNode);
            // pindahkan node sekarang ke
closed node
            if (curNode == endNode)
            { //sudah ketemu pathnya
                List<Node> path =
retracePath(startNode, endNode); //bangun
ulang pathnya
                //hentikan pencarian
                stopwatch.Stop();
                UnityEngine.Debug.Log("Wakt
u eksekusi A*: " +
stopwatch.ElapsedMilliseconds + " ms");
                seeker.GetComponent<SeekerM
ovement>().SetPath(path); //gerakkan
agentnya
                return;
            }

            foreach (Node neighbor in
grid.getNeighbors(curNode))
            { //periksa semua tetangga dari
node yang sekarang
                if (!neighbor.walkable ||
closedNode.Contains(neighbor))
                { //node tidak bisa
dilangkahi (adalah obstacle) atau nodenya
sudah closed
                    continue; //lewati
                }

                int costToNeighbor =
curNode.gCost + getDist(curNode, neighbor);
//hitung jarak dari titik ke neighbor yang
sekarang
                if (costToNeighbor <
neighbor.gCost
|| !openNode.Contains(neighbor))
                {
                    neighbor.gCost =
costToNeighbor; //ubah gCost neighbor
                    neighbor.hCost =
getDist(neighbor, endNode); //hitung
heuristik (jarak dari titik akhir ke
neighbor)
                    neighbor.parent =
curNode;

                    if
(!openNode.Contains(neighbor))
                    {
                        openNode.Add(neighb
or);
                    }
                }
            }
        }
        stopwatch.Stop();
        UnityEngine.Debug.Log("Waktu
eksekusi A*: " +
stopwatch.ElapsedMilliseconds + " ms");
    }
```

*Gambar 14: Source code algoritma A**
Sumber:Dokumentasi penulis

H. *Source Code Algoritma UCS*

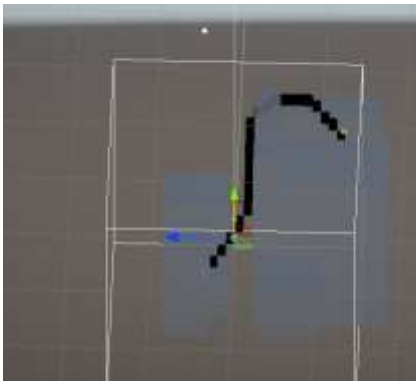
Berikut adalah source code untuk implementasi algoritma UCS.

```
void FindPath(Vector3 startPos, Vector3 endPos)
    {
        Stopwatch stopwatch = new Stopwatch();
        stopwatch.Start();
        //UCS
        Node startNode = grid.NodeFromWorldPoint(startPos); //node awal
        Node endNode = grid.NodeFromWorldPoint(endPos); //node tujuan

        List<Node> openNode = new List<Node>(); //node yang belum dikunjungi
        HashSet<Node> closedNode = new HashSet<Node>(); // node yang sudah dikunjungi
        openNode.Add(startNode);//masukkan ke open node

        while (openNode.Count > 0)
        {
            Node curNode = openNode[0]; //node sekarang

            for (int i = 1; i < openNode.Count; i++)
            {
                if (openNode[i].gCost < curNode.gCost)
                { // ganti node sekarang (UCS)
                    curNode = openNode[i];
                }
            }

            openNode.Remove(curNode);
            closedNode.Add(curNode);
            // pindahkan node sekarang ke closed node
            if (curNode == endNode)
            { //sudah ketemu pathnya
                List<Node> path = retracePath(startNode, endNode); //bangun ulang pathnya
                //hentikan pencarian
                stopwatch.Stop();
                UnityEngine.Debug.Log("Waktu eksekusi UCS: " + stopwatch.ElapsedMilliseconds + " ms");
                seeker.GetComponent<SeekerMovement>().SetPath(path); //gerakkan agentnya
                return;
            }

            foreach (Node neighbor in grid.getNeighbors(curNode))
            { //periksa semua tetangga dari node yang sekarang
                if (!neighbor.walkable || closedNode.Contains(neighbor))
                { //node tidak bisa dilangkahi (adalah obstacle) atau nodenya sudah closed
                    continue; //lewati
                }

                int costToNeighbor = curNode.gCost + getDist(curNode, neighbor); //hitung jarak dari titik ke neighbor yang sekarang
                if (costToNeighbor < neighbor.gCost
                || !openNode.Contains(neighbor))
                {
                    neighbor.gCost = costToNeighbor; //ubah gCost neighbor
                    neighbor.parent = curNode;

                    if (!openNode.Contains(neighbor))
                    {
                        openNode.Add(neighbor);
                    }
                }
```

```
            }
        }
        stopwatch.Stop();
        UnityEngine.Debug.Log("Waktu
eksekusi UCS: " +
stopwatch.ElapsedMilliseconds + " ms");
    }
```

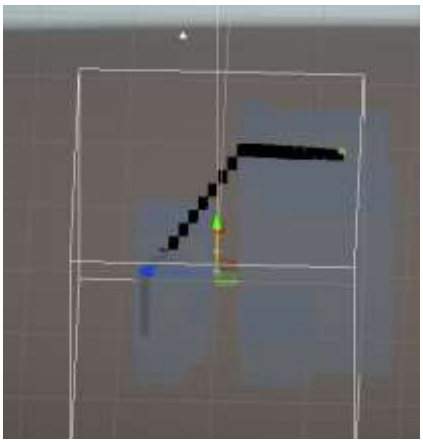
Gambar 15: Source code algoritma UCS
Sumber:Dokumentasi penulis

IV. Hasil Pengujian



*Gambar 16: Pengujian menggunakan algoritma A**

Sumber: Dokumentasi penulis



Gambar 17: Pengujian menggunakan Algoritma UCS

Sumber: Dokumentasi penulis

Berdasarkan path yang dipilih, dapat diperhatikan bahwa algoritma A* dan UCS sama-sama memilih path yang terpendek, mungkin ada perbedaan sedikit terhadap path yang dipilih. Namun, terdapat perbedaan terhadap peforma kedua algoritma berdasarkan waktu eksekusinya. (Lihat pranala video YouTube yang dilampirkan pada makalah ini untuk melihat simulasinya).

Gambar di bawah ini adalah waktu eksekusi dari algoritma A* dan UCS.

*Gambar 18: Performa Algoritma A**

Sumber: Dokumentasi Penulis

Gambar 19: Performa Algoritma UCS

Sumber: Dokumentasi penulis

Berdasarkan hasil waktu eksekusi di atas, A* memiliki waktu eksekusi yang jauh lebih rendah dari pada algoritma UCS. Algoritma A* memiliki rentang waktu eksekusi sekitar 326 ms – 0 ms, sedangkan UCS memiliki rentang waktu eksekusi sekitar 1381 ms – 0 ms.

V. KESIMPULAN

Berdasarkan hasil riset pada makalah ini, dapat disimpulkan bahwa algoritma yang lebih cocok digunakan untuk AI *pathfinding* untuk bot dalam gim tiga dimensi adalah algoritma A* karena algoritma A* memiliki waktu eksekusi yang lebih cepat dan peforma yang lebih baik dibandingkan dengan algoritma UCS (ataupun Dijkstra).

A* bisa bekerja lebih cepat daripada UCS karena algoritma A* menggunakan nilai heuristik yang memungkinkan bot untuk menghindari *node*/kubus yang dianggap tidak perlu dikunjungi karena costnya yang sudah dianggap terlalu besar, sedangkan algoritma UCS tidak menggunakan nilai heuristik, sehingga UCS menggunakan hampir seluruh *node*/kubus untuk mencari *path* terpendek.

Dengan alasan tersebut, algoritma A* lebih sering dipilih dan disukai untuk mengoptimisasi performa sebuah gim, terutama pada gim 3 dimensi. Bahkan, Unity menggunakan algoritma A* sebagai bagian dari *framework* UnityEngine.AI milik mereka.

PRANALA VIDEO YOUTUBE

Berikut ini adalah pranala video YouTube terhadap penjelasan isi dari makalah ini lebih lanjut:

https://www.youtube.com/watch?v=5qsEwaD-MUk

PRANALA REPOSITORI GITHUB

Berikut ini adalah pranala repositori github dari proyek ini.

https://github.com/DeltDev/3DBot-AStar-UCS

REFERENSI

[1] R. Munir, "IF2211 Strategi Algoritma - Semester II Tahun 2023/2024," [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/stima23-24.htm. [Diakses 12 Juni 2024].

[2] R. Munir, "IF2120 Matematika Diskrit – Semester I Tahun 2023/2024," [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/matdis23-24.htm. [Diakses 12 Juni 2024]

[3] S. Lague. *A* Pathfinding (E02: node grid)*. (18 Dec, 2014). Accessed: Jun 8, 2024. [Online video]. Available: https://www.youtube.com/watch?v=nhiFx28e7JY

[4] S. Lague. *A* Pathfinding (E03: algorithm implementation*). (20 Dec, 2014). Accessed: Jun 8,2024. [Online video]. Available: https://www.youtube.com/watch?v=mZfyt03LDH4

[5] A Bit Of Game Dev. *3D Pathfinding & Agent Avoidance In Unity*. (16 Nov, 2021). Accessed: Jun 9, 2024 [Online video]. Available: https://www.youtube.com/watch?v=p3WcsO6pAmU

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, 12 Juni 2024

Akbar Al Fattah
13522036