Optimisasi Algoritma Pathfinding Worker dengan Breadth First Search (BFS) dan Caching pada Permainan UnCiv

Muhammad Jordan Ferimeison - 13524047
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: ferimeison@gmail.com , 13524047@std.stei.itb.ac.id

Abstraksi—UnCiv merupakan salah satu permainan membangun peradaban dengan berbagai macam komponen. Salah satu komponen dalam game adalah kemampuan unit untuk menentukan Langkah secara otomatis. Salah satunya adalah unit worker. Unit ini bekerja secara otomatis dengan mencari petak di sekitarnya dan membangun improvement yang sesuai. Makalah ini akan membahas implementasi algoritma pathfinding pada worker dan optimasi yang mungkin dilakukan.

Keywords—BFS; UnCiv; Caching; Worker Unit on UnCiv; Civillian Unit on UnCiv;

I. PENDAHULUAN

Strategi algoritma merupakan salah satu mata kuliah yang didesign untuk memecahkan persoalan secara algoritmik []. Masalah yang diselesaikan dapat berupa masalah komputasi, masalah matematis, hingga masalah sehari-hari. Strategi algoritma mengajarkan bagaimana cara menyelesaikan suatu masalah secara sistematis dan terstruktur, dengan memahami pola dan menerapkan solusi permasalahan dengan sesuai (algoritma).

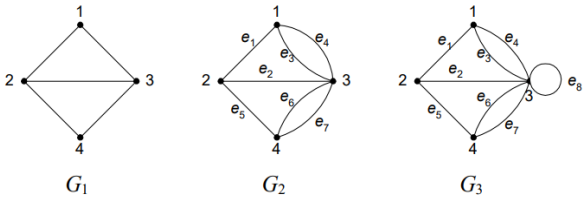
UnCiv merupakan permainan *remake* dari Civ V. Permainan ini dibangun oleh komunitas (cek link). Permainan ini mengajak pemain membangun peradaban (civilization) dengan berbagai aspek seperti science, culture, gold, happiness, policy, technology, city, population, espionage, war, trade, dan aspek lainnya untuk mendukung permainan. Permainan akan selesai ketika salah satu peradaban (civilization) mencapai syarat kemenangan. Dalam membangun peradaban, seorang pemain dapat menentukan keputusan sendiri, atau menyerahkan pengambilan keputusan kepada unitnya. Misalnya, dalam membangun improvement pada suatu petak, pemain dapat memilih improvement sendiri atau memilih opsi otomatis yang disediakan oleh permainan. Unit juga dapat bergerak sendiri ketika pemain memilih opsi otomasi. Selain itu, permainan juga menyediakan opsi otomasi lain seperti berperang, eksplorasi, membangun, menghubungkan kota, pembangunan oleh kota, dan masih banyak lagi.

Pada makalah ini, penulis mencoba salah satu *chore* pada permainan UnCiv. Aspek yang coba untuk dioptimasi adalah cara unit (khususnya *worker*) menemukan *path* secara otomatis ke suatu petak dengan algoritma Breadth First Search (BFS) dan *caching*. Algoritma ini memungkinkan komputasi yang dibutuhkan untuk menghitung jarak optimal *worker* ke suatu petak dengan mengakses memori (*cache*) jika pernah ditelusuri dan menelusuri path dengan BFS.

II. LANDASAN TEORI

A. *Graf*

Graf merupakan suatu diagram, cara, atau bahkan objek yang digunakan untuk merepresentasikan objek diskrit dan hubungan antarobjek tersebut [1]. Dalam merepresentasikan objek dan hubungannya, terdapat dua komponen utama dalam graf, yaitu *vertex* (simpul) dan *edge* (sisi). Suatu simpul biasanya merepresentasikan objek dan sisi merepresentasikan hubungan antar objek tersebut.



Gambar 1. Graf; Sumber: [1]

Graf dinotasikan secara umum sebagai G(V, E) dengan V sebagai simpul dan E sebagai sisi yang keduanya tidak boleh null (kosong) [1]. Graf juga dibagi menjadi beberapa jenis seperti,

1) Graf sederhana: graf yang tidak mengandung sisi ganda (ketika dua buah sisi menghubungkan dua simpul yang sama) dan tidak mengandung sisi gelang (ketika satu sisi menghubungkan satu simpul yang sama).

2) Graf tak sederhana: graf yang boleh mengandung sisi ganda maupun sisi gelang.

3) Graf tak berarah: graf yang sisinya tidak merepresentasikan hubungan berarah (dapat berlaku mutual).

4) Graf berarah: graf yang sisinya merepresentasikan hubungan berarah.

B. *Tree/Pohon*

Pohon merupakan graf tak berarah yang terhubung dan tidak memiliki siklus (*no loop*) [2]. Secara umum, pohon adalah graf yang memenuhi tiga syarat sebagai berikut [2],

1) Tidak berarah: tidak memiliki *edge* yang berarah

2) Terhubung: semua *vertex* terhubung paling tidak oleh satu *edge*

3) Tidak memiliki sirkuit (siklus): hanya terdapat satu path yang menghubungkan satu *vertex* dengan *vertex* lainnya.

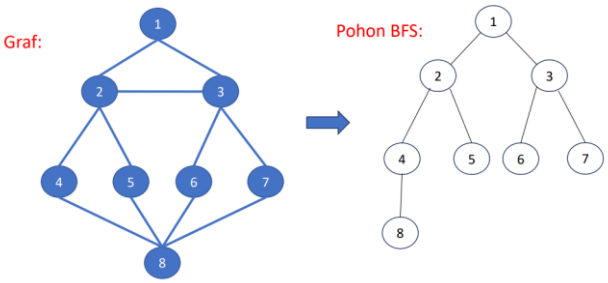
C. *Breadth First Search (BFS)*

Breadth First Search (BFS) merupakan algoritma traversal yang ditujukan untuk mencari rute dari suatu titik ke titik tujuan [3]. BFS mencari rute dengan menelusuri graf berdasarkan tingkat kedalamannya. Secara umum, BFS diimplementasikan sebagai berikut,

```
query[] ← first node
destination ← destination node
while (query not empty) do
   currNode ← query.pop()
   if ( currNode is destination ) then return true { exist route }
   foreach (child of currNode) do
      query.push(child)
return false { no route from first node to destination }
```

BFS merupakan algoritma traversal yang bersifat *uninformed search* [3]. Hal ini berarti algoritma BFS tidak memerlukan informasi mengenai seberapa jauh posisi relative suatu objek terhadap lokasi tujuan [3].

BFS mengubah graf menjadi pohon ketika melakukan algoritma tersebut seperti pada gambar berikut,



Gambar 2. Contoh Pohon BFS; Sumber: [1].

D. *Caching*

Caching merupakan salah satu cara bagi program untuk menyimpan data secara sementara. Penyimpanan ini dilakukan dengan tujuan mengurangi waktu tunggu dan membuat request di masa depan menjadi lebih cepat. Saat data ingin diakses dari cache, terdapat dua kemungkinan yang dapat terjadi. Pertama, cache hit. Cache hit menandakan bahwa data yang ingin dicari berada di dalam cache, sehingga proses komputasi untuk mencari hasil tidak perlu dilakukan (mengurangi kompleksitas menjadi O(1)). Kemungkinan kedua adalah cache miss, Ketika data yang ingin dicari tidak ditemukan pada cache [4]. Semakin banyak data yang dimuat dalam cache (untuk proses berulang), maka program akan berjalan lebih cepat karena mengurangi komputasi yang diperlukan.

E. *UnCiv*

Gambar 3. UnCiv *gameplay*; Sumber: UnCiv Github (link)

UnCiv merupakan permainan *remake* dari Civ V. Permainan ini mengajak pemain membangun peradaban (civilization) dengan berbagai aspek seperti *science*, *culture*, *gold*, *happiness*, *policy*, *technology*, *city*, *population*, *espionage*, *war*, *trade*, dan aspek lainnya untuk mendukung permainan. Permainan akan selesai ketika salah satu peradaban (*civilization*) mencapai syarat kemenangan. Pemain dapat membangun kota, melakukan riset teknologi, memilih *policy* untuk membantu perkembangan, berinteraksi dengan peradaban lain, hingga berperang dengan unit liar. Dalam membangun peradaban, seorang pemain dapat menentukan keputusan sendiri, atau menyerahkan pengambilan keputusan kepada unitnya. Misalnya, dalam membangun improvement pada suatu petak, pemain dapat memilih improvement sendiri atau memilih opsi otomatis yang disediakan oleh permainan. Unit juga dapat bergerak sendiri ketika pemain memilih opsi otomasi. Selain itu, permainan juga menyediakan opsi otomasi

lain seperti ketika berperang, eksplorasi, membangun, menghubungkan kota, pembangunan oleh kota, dan masih banyak lagi.

III. METODOLOGI

Pada permainan ini, setiap unit memiliki system otomasinya sendiri. Salah satunya adalah unit *worker*. Unit ini memungkinkan pemain untuk membangun *improvement* di petak. Worker dapat membangun improvement sesuai dengan jenis petak yang ditempati. Untuk mencapai petak tersebut, worker hanya dapat melewati petak dengan cost tetap yang sudah ditentukan. Suatu unit dapat bergerak ke petak tersebut dalam satu giliran jika cost yang dibutuhkan kurang dari movement yang dimiliki oleh unit (cek gambar 3).

Gambar 4. UnCiv *worker automation*; Sumber: arsip pribadi.

Petak pada permainan dapat dianalogikan sebagai *vertex*. Untuk setiap petak, terdapat 6 buah sisi yang memungkinkan perpindahan petak. Perpindahan ini dapat dinyatakan sebagai *edge*. Untuk berpindah dari suatu petak ke petak lain, suatu unit perlu melakukan pencarian dari *vertex* (petak) dia berada menuju ke *vertex* lain dengan beberapa restriksi. Terdapat restriksi movement cost. Setiap unit memiliki biaya untuk berpindah. Terdapat restriksi terrain, setiap unit tidak dapat melewati terrain tertentu atau memerlukan biaya tambahan untuk melewati terrain tertentu. Hal ini juga akan mempengaruhi unit *worker* ketika memutuskan untuk mengambil keputusan secara mandiri (*automation*). Ketika melakukan otomasi, worker perlu menemukan petak yang dapat ia tempuh untuk melakukan pergerakan. Setelah itu, worker akan meninjau nilai dari petak tersebut dan menyesuaikan pilihannya.

Di entry point findTileToWork, pergerakan worker diinisiasi dengan melakukan pengecekan pada setiap petak yang potensial.

```
            for           (tilePriorityGroup           in
workableTilesPrioritized) {

                var bestTile: Tile? = null

                for           (tileInGroup           in
tilePriorityGroup.value.sortedBy                        {
unit.getTile().aerialDistanceTo(it) }) {

                    //    These    are    the    expensive
calculations (tileCanBeImproved, canReach), so we only
apply these filters after everything else it done.

                    if    (!tileHasWorkToDo(tileInGroup,
unit)) continue

                    if  (unit.getTile()  ==  tileInGroup)
return unit.getTile()

                    if
(!unit.movement.canReach(tileInGroup)) continue

                    if    (bestTile    ==    null    ||
getFullPriority(tileInGroup,           unit)          >
getFullPriority(bestTile, unit)) {

                        bestTile = tileInGroup

                    }

                }

                if (bestTile != null) {

                    return bestTile

                }
```

```
            }
```

Sementara itu, UnitMovement.canReach diimplementasikan pada UnitMovement.kt sebagai,

```
fun canReach(destination: Tile) =
canReachCommon(destination)
{ getShortestPath(it).any() }
```

canReach() memanfaatkan fungsi canReachCommon() dan getShortestPath() untuk mengetahui apakah suatu petak dapat dicapai oleh *worker* unit. Fungsi canReachCommon() memanfaatkan kondisi unit saat ini dan kondisi petak untuk mengetahui secara umum apakah petak tersebut bisa dicapai atau tidak (dari kondisi ada unit di petak, unit sudah bergerak, dan hal lainnya). Sementara, getShortestPath melakukan komputasi untuk menemukan path dari unit meunuju petak. Fungsi kemudian menyimpan path dari unit ke petak ke dalam cache. Pencarian ini dilakukan berulang kali pada setiap petak yang dianggap valid oleh program. *Cache* yang disimpan memuat petak dan path menuju petak. Petak disimpan melalui kelas PathfindingCache. Kelas ini dipanggil ketika getShortestPath menemukan shortest path (hanya menyimpan petak). Pencarian memakan komputasi yang besar karena getShortestPath menghitung ulang path dari unit menuju petak secara terus menerus (karena tidak menyimpan path ke petak yang dilewati dari unit ke petak).

Hal ini juga tertera dalam WorkerAutomation.kt, lebih tepatnya pada baris 53, terdapat satu chore yang dijelaskan sebagai berikut "*UnitMovement.canReach still very expensive and could benefit from caching, it's not using BFS*." Artinya, saat ini UnitMovement.canReach menggunakan algoritma yang membutuhkan komputasi tinggi dan dapat memanfaatkan algoritma BFS dan *caching* untuk memperbaikinya.

Namun, mengapa BFS dan *Caching* dinilai mampu menjadi optimasi pada program yang diberikan? Pada algoritma pertama, anggap setiap petak yang dianggap *admissible* sebagai N, program akan memanggil fungsi canReach sebanyan N kali, dengan kompleksitas

$$O(N)$$

Untuk setiap fungsi, program akan mencari rute menuju ke petak tersebut. Anggap terdapat V buah petak. Untuk setiap canReach, akan dipanggil V buah operasi untuk mencari *path* (*worst case*), dengan kompleksitas

$$O(V)$$

Secara umum, algoritma pertama memiliki kompleksitas

$$O(N) \, . \, O(V) = O(N \, . \, V)$$

Pada solusi perbaikan, program akan mencoba melakukan pencarian dengan BFS untuk menemukan petak mana saja yang *admissible*. Jika menggunakan asumsi yang sama untuk N dan V, maka untuk setiap petak yang akan dicari, program hanya perlu melakukan 1 operasi untuk petak baru, atau dengan kompleksitas

$$O(1)$$

Karena terdapat N buah petak yang akan dicari, dan V buah petak total, maka dalam kondisi terbutuk akan dilakukan operasi sebanyak N + V operasi (V operasi untuk menelusuri semua petak dan N operasi untuk mencapai petak yang valid), atau dengan kompleksitas

$$O(N + V)$$

Misal, terdapat K buah worker yang melakukan operasi otomasi. Maka, untuk setiap worker akan dilakukan perhitungan mencari path. Maka, kompleksitas algoritma untuk mencari path untuk setiap worker dengan algoritma pertama (menggunakan pendekatan seperti program) adalah

$$O(K\ .\ (N\ .V))$$

dan untuk algoritm yang menggunakan BFS dan Cache adalah

$$O(K\ .(N + V))$$

Perubahan pendekatan ini mengurangi kompleksitas algoritma secara drastis. Program yang sebelumnya memerlukan $O(K\ .N\ .V)$ menjadi $O(K\ .(N + V))$.

IV. Implementasi dan Hasil

Untuk membuat *pathfinding worker* menjadi lebih efisien, kita perlu mengubah implementasi pada dokumen UnitMovement.kt. Ubah fungsi canReach() menjadi menggunakan pendekatan BFS dan tambahkan opsi *caching*. Pertama, tambahkan opsi *caching* untuk petak yang dicari dengan BFS pada kelas PathfindingCache (untuk diakses oleh fungsi canReach()) dengan menambahkan variable untuk menyimpan dan memperbarui *cache*.

```
// pada PathfindingCache
reachableTiles ← result from BFS, initialized as null { to store data, or as cache }

getReachableTiles() → BFS { function to update and get cached tiles }
begin.
  if ( reachableTiles is null or unitStateChanged ) then
    reachableTiles ← null{ clear cache }
    reachableTiles ← BFS(unit.getTile()) on canPassThrough(it)
  return reachableTiles
end.
```

Kode tersebut menggunakan fungsi BFS yang telah diimplementasikan dalam program. Berikut adalah implementasi BFS yang terdapat dalam program,

```
@InternalState
class BFS( val startingPoint: Tile,
    private val predicate : (Tile) -> Boolean ) {
    var maxSize = Int.MAX_VALUE
    private val tilesReached = HashMap<Tile, Tile>()
    init { tilesToCheck.add(startingPoint)
        tilesReached[startingPoint] = startingPoint }
    fun stepToEnd() { while (!hasEnded())nextStep() }
    fun stepUntilDestination(destination: Tile): BFS {
        while (!tilesReached.containsKey(destination) && !hasEnded()) nextStep()
        return this }
    fun nextStep(): Tile? {
        if (tilesReached.size >= maxSize) { tilesToCheck.clear(); return null }
        val current = tilesToCheck.removeFirstOrNull() ?: return null
        for (neighbor in current.neighbors) {
            if (neighbor !in tilesReached && predicate(neighbor)) { tilesReached[neighbor] = current
                tilesToCheck.add(neighbor) }
        }
        return current
    }
    @Readonly
    fun getPathTo(destination: Tile): Sequence<Tile> = sequence {
        var currentNode = destination
        while (true) {
            val parent = tilesReached[currentNode] ?: break  // destination is not in our path
            yield(currentNode)
            if (currentNode == startingPoint) break
            currentNode = parent
        }
    }
    fun hasEnded() = tilesToCheck.isEmpty()
    fun hasReachedTile(tile: Tile) = tilesReached.containsKey(tile)
    fun getReachedTiles(): MutableSet<Tile> = tilesReached.keys
    fun size() = tilesReached.size
}
```

Untuk menggunakan pendekatan BFS, kita perlu mengubah fungsi canReach dari menggunakan getShortestPath() menjadi implementasi BFS dengan memanfaatkan *caching* yang telah ditambahkan sebagai berikut,

```
canReach() → Boolean
begin.
```

```
    canReachCommon(destination) {
        bfs  ← getReachableTiles()
        if bfs.hasReachedTile(destination) then  return
true
        if !bfs.hasEnded()) then
            bfs.stepUntilDestination(destination)
            if  (bfs.hasReachedTile(destination))  then
return true
        if         (destination.neighbors.any         {
bfs.hasReachedTile(it)  }  and  canMoveTo(destination))
then
            return@canReachCommon true
        return false { cannot reach destination }
}
end.
```

Dibandingkan dengan memanggil fungsi yang mencari shortestPath ke petak destination, fungsi canReach mengubah pendekatan menjadi dengan menggunakan BFS. Program akan melakukan pengecekan dengan algoritma BFS apakah destinasi dapat dituju dari petak asal. Pendekatan ini mengurangi perhitungan path, yang sebelumnya harus menghitung ulang untuk setiap petak, menjadi menyimpan ke dalam cache yang baru diinisiasi pada PathfindingCache. Hal ini diharapkan meningkatkan *cache hit* pada program. Karena, program akan menyimpan petak yang dapat dituju dari titik awal (termasuk petak yang dibutuhkan untuk sampai ke titik tersebut). Sehingga,

Salah satu drawback yang dirasa dapat terjadi adalah terkait dengan variabel MAX_VALUE. Variabel ini menjadi pembatas bagi algoritma BFS (misal, jika *worker* ternyata dapat mencapai petak dengan jarak lebih dari int MAX_VALUE, maka akan terjadi kegagalan pada program). Namun, penulis belum dapat menjalankan perubahan secara langsung karena keterbatasan yang dimiliki. Oleh karena itu, kedepannya diharapkan implementasi dapat dijalankan secara langsung pada program (belum di run).

V. KESIMPULAN DAN SARAN

Penerapan algoritma BFS dalam menentukan petak yang dapat dijelajahi dan menyimpannya pada memori (cache) mengurangi waktu komputasi yang dibutuhkan dibandingkan dengan mencari path dari suatu petak ke unit secara berulang. BFS memungkinkan pencarian petak yang dapat dijelajahi dilakukan secara inklusif (memasikan setiap petak yang dapat dijelajahi ditandai dengan benar). Namun, implementasi ini meningkatkan konsumsi memori yang dibutuhkan oleh permainan. Permainan perlu mengalokasikan memori lebih untuk menyimpan petak yang dapat dikunjungi oleh worker. Saran optimasi kedepannya adalah dengan mempertimbangkan penggunaan memori dengan lebih efisien. Pendekatan ini (menggunakan BFS dengan memori) penulis nilai mirip dengan pendekatan program dinamis (*Dynamic Programming*). Pendekatan program dinamis memungkinkan program menyelesaikan masalah dengan memastikan langkah yang ditempuh selalu optimal. Langkap optimal ini kemudian disimpan pada memori agar komputasi tidak dilakukan berulang, dan jika ditemukan langkah lain yang lebih optimal, tidak perlu melakukan komputasi untuk perbandingan (hanya membandingkan nilai dengan cost O(1)). Namun, penulis merasa uncertain apakah implementasi ini dapat dikategorikan secara utuh menggunakan pendekatan program dinamis (karena lebih berfokus kepada pendekatan menggunakan BFS dibandingkan pemrograman dinamis yang memanfaatkan BFS).

Selain optimasi dengan menggunakan BFS dan Caching, masih terdapat chore lain yang dapat diimplementasi dengan pendekatan lain. Misalnya, chore pada City untuk melakukan optimasi road path (koneksi antar City) dengan menggunakan multi-target A* hingga memanfaatkan pemrograman dinamis untuk helper dalam menentukan pergerakan paling optimal (untuk player, *knapsack-like* problem dengan parameter science, culture, gold, etc). Masih banyak hal yang sebenarnya dapat dilakukan (terkait strategi algoritma) kepada permainan ini. Selain itu, banyak aspek dalam permainan yang telah diimplementasi menggunakan konsep yang ada dalam strategi algoritma (tetapi tidak dapat penulis jabarkan satu persatu). Namun, karena keterbatasan penulis, penulis hanya melakukan analisis dan upaya dalam menyelesaikan *chore* yang terdapat pada permainan. Harapannya, implementasi tersebut dapat diterapkan kedepannya untuk meningkatkan kualitas permainan dan memberikan kontribusi kepada komunitas.

LAMPIRAN

Lampiran 1.

Github UnCiv: https://github.com/yairm210/Unciv

ACKNOWLEDGMENT

Penulis berterima kasih kepada Tuhan Yang Maha Esa karena telah mengizinkan penulis membuat makalah ini. Penulis berterima kasih kepada Dr. Ir. Rinaldi Munir, M.T. sebagai dosen pengampu IF2211 K1. Penulis juga berterima kasih kepada Bhisma Sutan Danusiri dan Suryaputra Rasyid P. yang telah merekomendasikan permainan ini kepada penulis. Selain itu, penulis juga ingin mengucapkan terima kasih kepada komunitas UnCiv yang telah membuat dokumentasi dan program permainan sebagai program *open-source*.

REFERENCES

[1] R. Munir, Matematika Diskrit. Website. 19 Juni 2026. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

[2] R. Munir, Matematika Diskrit. Website. 19 Juni 2026. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/23-Pohon-Bag1-2024.pdf

[3] R. Munir, Strategi Algoritma. Website. 19 Juni 2026. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/01-Pengantar-Strategi-Algoritma-(2026).pdf

[4] Zhong, Liang; Zheng, Xueqian; Liu, Yong; Wang, Mengting; Cao, Yang (February 2020). "Cache hit ratio maximization in device-to-device communications overlaying cellular networks". China Communications. 17 (2): 232–238. Bibcode:2020CComm..17b.232Z. doi:10.23919/jcc.2020.02.018. ISSN 1673-5447. S2CID 212649328.

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, 19 Juni 2026

Muhammad Jordan Ferimeison 13524047