Perbandingan Algoritma UCS dan A* untuk Pencarian Solusi pada Permainan Sokoban

Aryo Bama Wiratama – 13523088[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: aryobama09@gmail.com , 13523088@std.stei.itb.ac.id

***Abstract*—Permainan Sokoban merupakan permainan teka-teki klasik yang mengharuskan pemain memindahkan kotak ke lokasi target yang telah ditentukan. Makalah ini mengeksplorasi dan menganalisis kinerja dan membandingkan dua algoritma pencarian jalur, yaitu Uniform Cost Search (UCS) dan A* (A-star) untuk menemukan solusi optimal (jumlah langkah minimum) dalam permainan Sokoban. Pemetaan masalah dilakukan dengan merepresentasikan setiap konfigurasi papan (posisi pemain dan semua kotak) sebagai sebuah *state* dalam ruang pencarian. Implementasi algoritma dilengkapi dengan fungsi heuristik berbasis Manhattan Distance untuk A* dan deteksi *deadlock* sebagai *bounding function* untuk kedua algoritma. Pengujian dilakukan pada beberapa kasus untuk membandingkan waktu eksekusi dan jumlah simpul yang dikunjungi oleh kedua algoritma. Hasil pengujian menunjukkan bahwa UCS memiliki waktu eksekusi yang lebih cepat meskipun A* mengunjungi simpul yang lebih sedikit yang mengindikasikan bahwa fungsi heuristik Manhattan Distance yang digunakan tidak cukup efisien dalam membantu algoritma A*.**

Keywords—sokoban, uniform cost dearch, A*, manhattan distance, Deadlock, fungsi heuristik

I. PENDAHULUAN

Sokoban adalah salah satu permainan teka – teki klasik yang berasal dari Jepang. Sokoban dikembangkan pada tahun 1982 oleh Hiroyuki Imabayashi. Kata *Sokoban* sendiri berasal dari kosa kata jepang yang artinya "penjaga gudang". Sesuai namanya, pada permainan ini pemain akan berperan sebagai penjaga gudang. Pemain dapat mengendalikan seorang karakter yang berada di dalam gudang dan bertugas menempatkan kotak-kotak sedemikian rupa sehingga berada di lokasi yang telah ditentukan.

Gambar 1.1 Permainan Sokoban
(Sumber: https://play.google.com/store/apps/details?id=com.gamesbykevin.sokoban&hl=id)

Walaupun terlihat sederhana, permainan ini memiliki kesulitan yang cukup tinggi. Pemain dituntut untuk berpikir logis dan strategis. Pemain harus memperhitungkan setiap langkahnya dengan cermat karena kotak hanya dapat didorong dan tidak bisa ditarik. Kesalahan perhitungan dalam menentukan urutan gerakan kotak dapat menyebabkan kotak - kotak berada di posisi yang tidak ideal dan dapat menyebabkan *deadlock* sehingga level tidak dapat diselesaikan dan pemain harus mengulang *progress*-nya dari awal.

Aturan permainan yang mudah dimengerti dan tingkat kesulitan yang meningkat tiap levelnya membuat permainan ini memiliki daya tarik tersendiri. Permasalahan utama permainan ini tidak hanya terletak pada cara memindahkan kotak ke tempat tujuan, tetapi juga bagaimana cara menentukan urutan pemindahan kotak sedemikian rupa sehingga solusi yang dihasilkan efektif dan efisien. Penyelesaian Sokoban melibatkan kombinasi pemikiran logis, strategi perencanaan jangka panjang, dan kemampuan dalam menganalisis pola gerakan. Hal ini menjadikan Sokoban sebagai contoh yang menarik dalam studi *pathfinding* atau pencarian jalur tercepat.

Terdapat beberapa algoritma yang dapat digunakan untuk melakukan pencarian jalur dalam permainan sokoban. Makalah ini berfokus pada eksplorasi beberapa algoritma pencarian jalur dalam Sokoban, yaitu UCS dan A* (star). Penulis akan membahas prinsip kerja masing – masing algoritma, penggunaannya dalam skenario permainan, pengukuran kinerja dari kedua algoritma, dan mengimplementasikan algoritma dalam bahasa Java. Dengan pendekatan ini, diharapkan makalah dapat memberikan gambaran menyeluruh mengenai kelebihan dan keterbatasan masing-masing algoritma dalam menangani permasalahan spasial dan perencanaan gerakan seperti yang terdapat pada permainan Sokoban.

II. LANDASAN TEORI

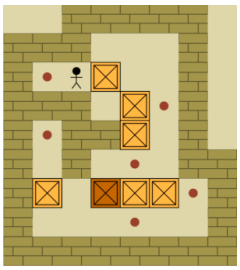
A. *Sokoban*

Sokoban adalah permainan teka-teki logika di mana pemain bertujuan untuk meletakkan kotak-kotak ke lokasi yang sudah ditentukan. Pemain mengendalikan seorang karakter yang hanya dapat mendorong satu kotak dalam satu waktu dan tidak

diperbolehkan menarik kotak. Permainan ini biasanya disajikan dalam bentuk grid dua dimensi dengan komponen-komponen, seperti dinding, lantaim kotak, dan target lokasi.

Permainan ini terdiri dari beberapa komponen utama, yaitu:

1. Papan

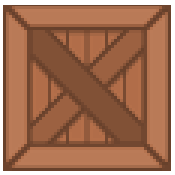


Gambar 2.1 Papan permainan Sokoban

(Sumber: https://user-images.githubusercontent.com/2433219/104414591-c449eb00-552d-11eb-8454-4a631f3f5be2.png)

Papan permainan dalam Sokoban merupakan tempat pemain berinteraksi dan menyelesaikan teka-teki. Papan ini biasanya berbentuk grid dua dimensi yang terdiri atas berbagai sel. Terdapat dua komponen utama yang membentuk papan, yaitu lantai, dan dinding. Lantai merupakan sel dasar yang dapat dilalui oleh pemain ataupun kotak. Pemain dapat bergerak bebas di atas lantai dan kotak dapat didorong di sel ini. Dinding berfungsi sebagai penghalang yang tidak dapat dilalui oleh pemain ataupun kotak. Sel ini menjadi batasan yang dapat memengaruhi startegi gerakan

2. Kotak

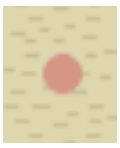


Gambar 2.2 Kotak pada permainan Sokoban

(Sumber: https://play.google.com/store/apps/details?id=br.com.orangevoid.sokobanfree)

Kotak merupakan objek yang harus dipindahkan ke lokasi tertentu pada lantai. Pemain hanya dapat mendorong satu kotak dalam satu waktu dan tidak dapat menerik kotak.

3. Target



Gambar 2.3 Target kotak

(Sumber: https://www.mathsisfun.com/games/images/sokoban.png)

Target merupakan lokasi yang harus ditempati oleh kotak agar permainan dinyatakan selesai. Biasanya ditandai dengan simbol khusus. Apabila di atasnya diletakkan sebuah kotak, itu akan disebut sebagai *box on goal.*

4. Pemain

Gambar 2.4 Karakter pada Sokoban

(Sumber: https://www.mathsisfun.com/games/images/sokoban.png)

Karakter yang dikendalikan oleh pemain untuk melakukan aksi dorongan terhadap kotak. Pemain dapat melakukan gerakan secara horizontal dan vertikal. Posisi awal pemain ditentukan dalam tiap level dan dapat memengaruhi strategi permainan.

B. Masalah Pencarian Jalur (Pathfinding Problem)

Masalah pencarian jalur merupakan salah satu permasalahan mendasar dalam bidang ilmu komputer. Masalah ini membahas tentang bagaimana cara menemukan suatu rute atau urutan langkah terbaik dari satu titik awal ke satu titik tujan dalam suatu ruang pencarian yang didefinisikan dan terdiri dari serangkaian kemungkinan keadaan.

Secara umum, suatu ruang pencarian dapat direpresentasikan dalam bentuk graf. Dalam representasi graf, jalur atau lintasan atau rute merupakan urutan simpul yang dihubungkan oleh sisi pada graf. Simpul pada graf merepresentasikan suatu suatu keadaan atau posisi dan sisi pada graf merepresentasikan suatu kemungkinan perpindahan antar simpul. Biasanya graf yang digunakan untuk merepresentasikan suatu jalur adalah graf berbobot dengan nilai atau bobot merepresentasikan waktu, jarak tempuh atau sumber daya yang diperlukan untuk berpindah dari satu simpul ke simpul yang lain. Dalam graf, posisi awal disebut dengan *start node* dan posisi tujuan disebut dengan *goal node*.

Jalur terbaik dalam masalah pencarian jalur memiliki definisi yang berbeda-beda tergantung konteks masalah. Terdapat beberapa kriteria yang sering digunakan untuk mendefinisikan jalur terbaik, yaitu

1. Jalur terpendek: Meminimalkan jarak fisik atau jumlah langkah yang diambil.
2. Jalur tercepat: Meminimalkan waktu yang harus ditempuh
3. Jalur termurah: Meminimalkan risiko atau bahaya yang mungkin ditemui di sepanjang jalur.

C. Fungsi Heuristik

Heuristic merupakan kata yang diambil dari bahasa Yunani yang secara harfiah berarti “mencari” atau “menemukan”. Dalam konteks pencarian jalur, fungsi heuristik adalah sebuah metode yang digunakan untuk menghitung estimasi biaya yang dibutuhkan dari keadaan/posisi saat ini hingga tujuan akhir. Tujuan dari heuristik adalah membantu algoritma dalam

mengambil keputusan yang lebih efisien dengan memprioritaskan langkah yang dianggap menjanjikan.

Berdasarkan sifat estimasinya, heuristik dibagi menjadi dua, yaitu

1. *Admissible heuristics*

   Fungsi heuristik dikatakan *admissible* jika fungsi tidak pernah melebih-lebihkan estimasi biaya yang dibutuhkan untuk mencapai tujuan. Artinya nilai dari fungsi heuristik selalu lebih kecil atau sama dengan biaya sesungguhnya terpendek ke tujuan

2. *Non-Admissible heuristics*

   Sebaliknya, fungsi heuristik dikatakan non-admissible jika dapat melebih-lebihkan biaya estimasi yang dibutuhkan untuk mencapai tujuan. Artinya nilai dari fungsi heuristik akan lebih besar daripada biaya sesungguhnya yang dibutuhkan.

D. Algoritma Uniform Cost Search (UCS)

Algoritma *Uniform Cost Search* (UCS) merupakan algoritma penentuan rute (*route planning*) yang digunakan untuk mencari jalur dengan biaya terendah dari suatu titik awal (*start node)* menuju satu titik tujuan (*goal node)*. Dalam konteks algoritma UCS, biaya dapat berupa jarak, waktu, dan hal lainnya yang berhubungan dengan *resource* yang dibutuhkan untuk berpindah posisi/*state*.

Algoritma ini masuk ke dalam kategori *uniformed search* karena dalam proses pencariannya tidak memanfaatkan informasi tambahan (seperti estimasi jarak ke tujuan) untuk membimbing arah pencarian. UCS hanya bergantung pada biaya kumulatif dari simpul awal menuju simpul saat ini untuk menentukan simpul mana yang akan dibangkitkan terlebih dahulu.

Algoritma ini memanfaatkan struktur data *priority queue* (antrian prioritas) untuk menyimpan simpul-simpul yang akan dibangkitkan.

Berikut cara kerja algoritma UCS secara umum

1. Inisialisasi:
   - Buat *priority queue* dengan kondisi kosong
   - Masukkan simpul awal ke dalam *queue* dengan biaya awal (g(n)) 0
2. Selama antrian belum kosong
   - Ambil simpul dengan biaya terendah dari *queue*
   - Periksa apakah simpul tersebut adalah simpul tujuan. Jika iya, algoritma ini selesai dan telah ditemukan jalur dengan biaya minimum. Jika tidak, lanjutkan proses pencarian
   - Jika simpul belum dikunjungi:
     - Tandai sebagai dikunjungi
     - Perluas simpul dengan mencari semua simpul tetangganya
     - Untuk setiap simpul tetangga, tentukan biaya yang dibutuhkan untuk menuju simpul (g(n)) tersebut.
     - Jika simpul tetangga belum pernah dikunjungi sebelumnya, masukkan simpul tetangga ke dalam *queue*
   - Ulangi langkah di atas hingga *queue* kosong atau ada solusi yang ditemukan

E. Algoritma A Star (A)*

Algoritma A* (dibaca A star) juga merupakan salah satu algoritma rute (*route planning*). Pertama kali ditemukan pada tahun 1968 oleh Peter Hart, Nils Nilsson, dan Betram Raphael di Stanford Research Institute (sekarang SRI International). Algoritma A* merupakan pengembangan dari algoritma pencarian sebelum A*, yaitu algoritma best first search (BFS) dan algoritma Dijkstra. A* star menggabungkan dua konsep utama pada Dijkstra dan BFS, yaitu biaya yang telah dikeluarkan dari sumber ke simpul n (g(n)) dan perkiraan biaya dari simpul n menuju simpul tujuan yang biasanya disebut fungsi heuristik (h(n)). Algoritma A* mengevaluasi kedua biaya tersebut untuk menentukan simpul mana yang akan dilalui.

Algoritma ini masuk ke dalam kategori *informed search* karena proses pencarian dibantu oleh informasi yang didapat dari fungsi heursitik.

Algoritma ini juga memanfaatkan struktur data *priority queue* untuk menyimpan simpul-simpul yang akan dibangkitkan. Cara kerja algoritma A* secara umum hampir sama dengan algoritma UCS. Perbedaannya adalaha pada A* prioritas bukan hanya ditentukan oleh fungsi biaya, melainkan juga oleh fungis heuristik

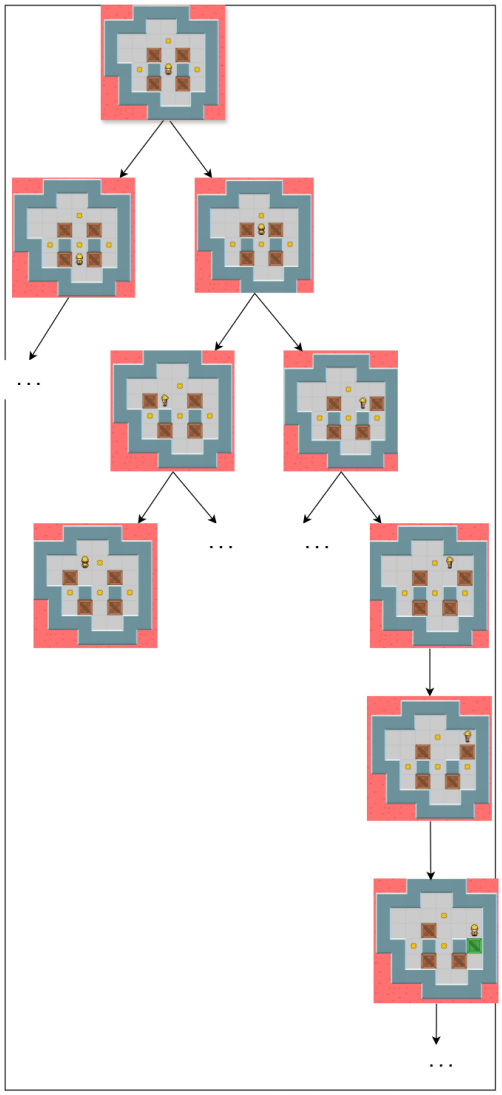
III. PEMETAAN MASALAH DAN IMPLEMENTASI

Tujuan dari permainan Sokoban adalah bagaimana cara memindahkan semua kotak ke lokasi yang sudah ditentukan dengan langkah sesedikit mungkin. Algoritma *pathfinding* dapat digunakan dalam menentukan solusi dalam permainan Sokoban. Dalam makalah ini, akan digunakan dua algoritma *pathfinding*, yaitu algoritma *uniform cost search* dan algoritma A Star. Untuk menggunakan kedua algoritma di atas, akan dilakukan pemetaan komponen permainan Sokoban ke dalam elemen-elemen permasalahan pencarian.

A. Ruang Pencarian

Dalam permainan Sokoban, ruang pencarian merpresentasikan seluruh kemungkinan langkah yang dapat dicapai dari kondisi awal hingga kondisi tujuan. Definisi *state* dalam ruang pencarian digambarkan oleh posisi pemain dan seluruh kotak pada papan permianan. Setiap posisi dalam papan dapat direpresentasikan sebagai koordinat (x, y). Setiap aksi yang dilakukan pemain, seperti bergerak atau mendorong kotak akan menghasilkan satu *state* Kompleksitas ruang pencarian sangat tinggi karena terdapat banyak kemungkinan posisi kotak dan pemain.

Kondisi awal dari permainan mencakup posisi awal pemain dan posisi seluruh kotak pada papan. Posisi ini diletakkan secara acak tergantung level permianan. Sedangkan kondisi tujuan adalah kondisi di mana seluruh kotak berhasil dipindahkan ke lokasi yang telah ditentukan.



Gambar 3.1 Ilustrasi Pohon Ruang Status dalam Permainan Sokoban

(Sumber: Arsip penulis)

B. *Aksi Pemain*

Pemain dapat melakukan aksi berupa pergerakan ke empat arah utama, yaitu atas, bawah, kiri, dan kanan. Aksi ini dapat dibedakan menjadi dua jenis, yaitu pergerakan biasa ke sel kosong, dan aksi mendorong kotak apabila terdapat kotak di arah tujuan dan di belakang kotak tersebut terdapat sel kosong yang dapat ditempati. Setiap aksi yang dilakukan pemain akan mengubah state permainan. Namun, aksi hanya dianggap valid jika memenuhi syarat tertentu, misalnya pemain tidak dapat menembus dinding, tidak dapat mendorong lebih dari satu kotak sekaligus, dan tidak dapat mendorong kotak ke luar batas papan atau ke posisi yang sudah terhalang.

Setiap aksi yang valid akan menghasilkan sebuah state baru yang menjadi anak dari state sebelumnya. Dalam implementasi algoritma pencarian, aksi-aksi ini digunakan untuk menelusuri ruang pencarian, dengan membangkitkan simpul-simpul baru dari simpul saat ini.

C. *Simpul*

Simpul dalam akan permainan Sokban akan merepresentasikan setiap kondisi papan. Simpul tidak hanya merepresentasikan sebuah kondisi, tetapi juga menyimpan informasi yang diperlukan dalam algoritma *pathfinding*.

Sebuah simpul dalam implementasi Sokoban menyimpan beberapa informasi, yaitu:

1. *State*: Konfigurasi pemain dan semua kotak pada papan permainan
2. Biaya (g(n)): Biaya kumulatif yang dibutuhkan untuk mencapai simpul tersebut dari simpul awal
3. Fungsi Heuristik (h(n)): Fungsi yang digunakan untuk menghitung estimasi biaya yang dibutuhkan dari kondisi saat ini hingga kondisi akhir. Digunakan untuk algoritma A Star
4. *Parent node*: simpul sebelumnya yang membangkitkan simpul saat ini. Digunakan untuk merekonstruksi jalur jika solusi telah ditemukan.
5. Aksi yang diambil: Arah gerakan pemain yang dilakukan pemain dari simpul induk ke simpul saat ini.

D. *Fungsi Biaya*

Fungsi biaya digunakan untuk mengukur seberapa besar "harga" yang harus dibayar untuk mencapai suatu *state* dari *initial state*. Dalam konteks Sokoban, fungsi biaya adalah jumlah langkah yang telah dilakukan oleh pemain. Setiap aksi valid yang dilakukan akan dikenakan biaya sebesar satu satuan langkah. Dengan demikian, total biaya suatu simpul adalah jumlah seluruh aksi dari awal permainan hingga posisi saat ini. Fungsi biaya biasanya dilambangkan dengan g(n)

E. *Fungsi Heuristik*

Fungsi heuristik akan digunakan untuk algoritma A star. Fungsi ini akan membantu algoritma dalam menulusui jalur. Dengan fungsi heuristik yang tepat, algoritma tidak akan menulusuri simpul yang tidak releven atau jauh dari solusi sehingga proses pencarian akan menjadi lebih efisien. Fungsi heuristik biasanya dilambangkan dengan h(n)

Berikut fungsi heuristik yang akan diimplementasikan dalam permainan Sokoban.

1. Fungsi heuristik berbasis manhattan distance

   Fungsi heuristik yang digunakan dalam implementasi ini akan didasarkan pada manhattan distance. Fungsi ini akan menghitung estimasi jarak total antara posisi kotak dengan posisi tujuan. Setiap kotak akan dipasangkan dengan target terdekat yang belum terpakai. Fungsi ini menggunakan pendekatan algoritma *greedy* untuk memasangkan tiap kotak dengan target. Fungsi heuristik ini memenuhi sifat *admissible* karena tidak melebih-lebihkan biaya sebenarnya untuk mencapai goal.
2. Pendeteksi *deadlock*

   Dalam permainan Sokoban, pemain dapat berhadapan dengan kondisi *deadlock*. *Deadlocck* merupakan kondisi di mana satu atau lebih kotak tidak lagi dapat dipindahkan ke posisi tujuan sehingga solusi tidak mungkin dicapai. Oleh karena itu, penulis mengimplementasikan fungsi yang dapat mendeteksi kondisi *deadlock* dalam papan permainan. Tujuannya agar algoritma tidak menulusuri simpul yang

tidak akan pernah mencapai solusi sehingga algoritma pencarian akan menjadi lebih efisiein. Fungsi ini sebenarnya bukan merupakan fungsi heuristik, tetapi lebih mendekati ke fungsi pembaatas atau *bounding function* pada algoritma *branch and bound* yang tujuan adalah membunuh simpul yang tidak akan mencapai solusi.

F. Fungsi Evaluasi

Kedua algoritma menggunakan struktur data *priority queue* untuk menyimpan simpul. Simpul dengan prioritas tertinggi akan dipilih untuk menjadi simpul ekspan. Penentuan prioritas pada kedua algoritma akan ditentukann berdasarkan nilai fungsi evaluasi. Semakin kecil nilai fungsi evaluasi maka semakin tinggi prioritasnya. Fungsi evaluasi biasanya disimbolkan dengan f(n). Fungsi evaluasi pada tiap algoritma mempunyai cara perhitungan yang berbeda-beda.

Berikut perhitungan fungsi evaluasi pada tiap algoritma:

1. Algoritma Uniform Cost Search

$$f(n) = g(n)$$

Di mana g(n) merupakan fungsi biaya

2. Algoritma A Star

$$f(n) = g(n) + h(n)$$

Di mana g(n) merupakan fungsi biaya dan h(n) merupakan fungsi heuristik

G. Implementasi Algoritma

Kedua algoritma memiliki cara kerja yang identik. Perbedaan dari kedua algoritma hanyalah terletak pada perhitungan fungsi evaluasinya. Pada A star, perhitugan fungsi evaluasi mempertimbangkan fungsi biaya dan fungsi heuristik, sedangkan UCS hanya mempertimbangkan fungsi biaya saja. Berikut adalah tahapan penyelesaian permainan Sokoban secara garis besar dengan menggunakan kedua algoritma:

1. Inisialisasi Awal
   - Program menerima input berupa objek papan permainan
   - Input tersebut akan memuat posisi awal pemain, posisi kotak, dan posisi target
   - Dibuat *priority queue* untuk menyimpan antrian simpul. Prioritas akan bergantung pada nilai fungsi evaluasi f(n) di mana sudah dijelaskan cara penentuan fungsi evaluasi untuk kedua algorirma
   - Dibuat sebuah *set* untuk menyimpan simpul yang sudah dikunjungi, disebut dengan *visitedStates*
2. Memasukkan simpul awal ke *queue*
   - Simpul awal yang berasal dari input pemain akan dibuat dengan nilai g(n) = 0 (artinya belum ada langkah yang dijalankan)
   - Simpul ini kemudian dimasukkan ke *visitedStates* untuk menandai bahwa telah dikunjungi
3. Proses Pencarian
   - Algoritma akan terus mengambil simpul dengan f(n) terkecil dan memprosesnya.
   - Akan diperiksa simpul yang sedang diproses (*currentNode*) apakah simpul tersebut adalah simpul tujuan.
4. Ekspansi simpul
   - Jika bukan simpul tujuan, algoritma akan mencoba semua kemungkinan arah gerakan. Untuk tiap gerakan akan menghasilkan *state* baru yang akan menjadi simpul anak.
   - Jika gerakan menyebabkan *deadlock* simpul tersebut akan diabaikan
   - Jika simpul valid dan belum pernah dikunjungi, akan dibuat simpul baru dengan g(n) = g(n) + 1. Pada A Star, akan dihitung nilai fungsi heuristiknya untuk state tersebut. Kemudian simpul akan dimasukkan ke *queue* dan dicatat ke *visitedStates*

Berikut adalah implementasi langkah di atas dalam bentuk pseudocode:

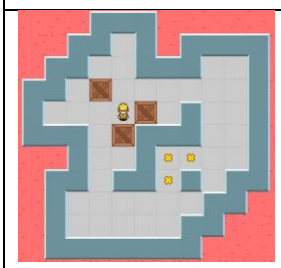
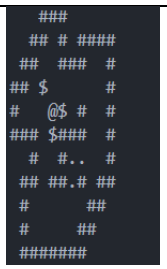
```
FUNCTION FindSolution(initialBoard):
  initialState ← GameState(initialBoard)
  frontier ← PriorityQueue ordered by gCost
  visitedStates ← empty Set
  initialNode ← Node(state=initialState, parent=null,
move=null, gCost=0)
  frontier.add(initialNode)
  visitedStates.add(initialState)

  WHILE frontier is not empty:
    currentNode ← frontier.poll()  // Ambil node dengan
biaya terendah
    currentState ← currentNode.state

    IF currentState is goal:
      lastFoundGoalNode ← currentNode
      RETURN ReconstructPath(currentNode)

    FOR each direction IN [UP, DOWN, LEFT, RIGHT]:
      result ← TryMove(currentState, direction)

      IF result ≠ null:
        nextState ← result.nextState
        move ← result.move

        IF DeadlockDetector.isDeadlock(nextState):
          CONTINUE

        IF nextState not in visitedStates:
          nextNode ← Node(state=nextState,
parent=currentNode, move=move,
gCost=currentNode.gCost + 1)
          frontier.add(nextNode)
          visitedStates.add(nextState)

  RETURN null // Tidak ditemukan solusi
```

```
FUNCTION TryMove(currentState, direction):
  nextPlayerPos ← posisi pemain setelah bergerak ke arah 'direction'

  IF nextPlayerPos adalah dinding:
    RETURN null

  IF nextPlayerPos berisi kotak:
    nextBoxPos ← posisi kotak setelah terdorong

    IF nextBoxPos adalah dinding ATAU berisi kotak lain:
      RETURN null

    newBoxPositions ← update posisi kotak setelah didorong
    newState ← GameState dengan posisi pemain dan kotak baru
    RETURN MoveResult(newState, Move(direction, push=True))
  ELSE:
    newState ← GameState dengan pemain pindah tanpa mendorong kotak
  RETURN MoveResult(newState, Move(direction, push=False))
```

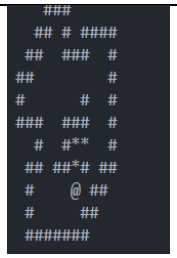
IV. Pengujian dan Analisis

Untuk melakukan pengujian, dibuat sebuah program dalam bahasa java yang mengimplementasikan semua komponen dan algoritma yang akan diperlukan. Kemudian berdasarkan program yang telah dibuat, akan dilakukan pengujian dengan kasus – kasus berikut:

1. Kasus 1

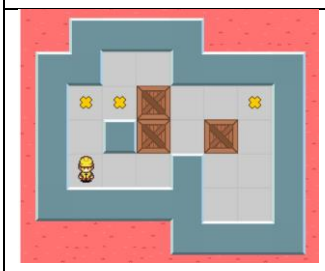
Input	Representasi dalam matrix
	

Tabel 4.1.1 Tabel Input Kasus 1

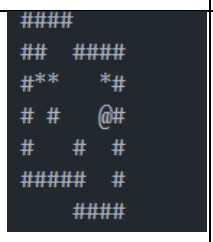
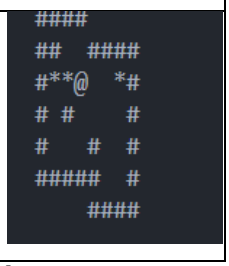
Algoritma	UCS	A*
Node Visited	183292	182593
Total Langkah	134	134
Waktu Eksekusi (ms)	3748	5492
Matriks Solusi		

Tabel 4.1.2 Tabel Hasil Kasus 1

2. Kasus 2

Input	Representasi dalam matrix
	

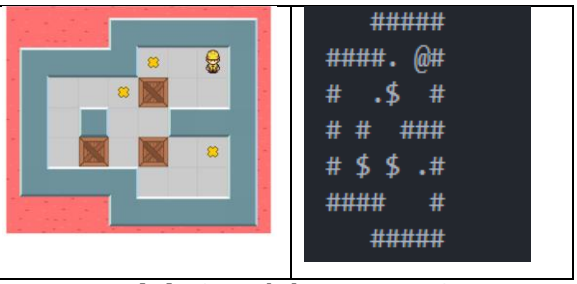
Tabel 4.2.1 Tabel Input Kasus 2

Algoritma	UCS	A*
Node Visited	1232	1183
Total Langkah	58	58
Waktu Eksekusi (ms)	4	7
Matriks Solusi		

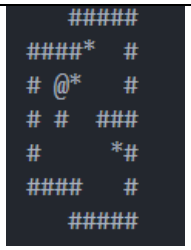
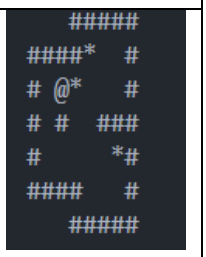
Tabel 4.2.2 Tabel Hasil Kasus 1

3. Kasus 3

Input	Representasi dalam matrix



Tabel 4.3.1 Tabel Input Kasus 3

Algoritma	UCS	A*
Node Visited	2069	1980
Total Langkah	51	51
Waktu Eksekusi (ms)	16	17
Matriks Solusi		

Tabel 4.3.2 Tabel Hasil Kasus 1

Berdasarkan pengujian 3 kasus berbeda, didapatkan bahwa waktu eksekusi algoritma UCS lebih cepat daripada A*. Hal ini menandakan bahwa dalam penyelesaian permainan Sokoban, UCS lebih baik secara waktu eksekusi daripada A*. Namun perhatikan bahwa simpul yang dikunjungi oleh algoritma A* lebih sedikit daripada algoritma UCS.

Secara logika, semakin sedikit jumlah simpul yang dikunjungi maka semakin cepat juga waktu eksekusinya. Namun, pengujian di atas membantah argumen tersebut. Terlihat bahwa walaupun A Star memiliki jumlah simpul terkunjungi yang lebih sedikit, tidak dijamin bahwa waktu eksekusinya lebih cepat. Hal ini dipengaruhi oleh pemilihan fungsi heuristik pada algoritma A star. Apabila fungsi heuristik yang digunakan cukup kompleks dan memiliki kompleksitas yang tinggi, fungsi heuristik justru akan menghasilkan waktu eksekusi yang lebih lama. Perhatikan pula bahwa jumlah node yang dikunjungi tidak jauh berbeda antara A* dan UCS. Hal ini menunjukkan bahwa heuristik yang digunakan tidak membantu algoritma A* secara signifikan dalam memilih rute. Pada makalah ini fungsi heuristik yang digunakan adalah pasangan kotak dengan target berdasarkan manhattan distance. Dari pengujian di atas terbukti bahwa fungsi heuristik yang dipilih kurang efisien dan tidak cukup membantu.

V. KESIMPULAN

Berdasarkan analisis terhadap kasus-kasus yang telah dilakukan pengujian, didapatkan kesimpulan bahwa algoritma UCS adalah algoritma yang dapat memberikan waktu eksekusi lebih cepat daripada A star dalam penyelesaian masalah Sokoban. Didapatkan juga bahwa fungsi heuristik yang dipilih pada makalah ini tidak cukup efisien dalam membantu algoritma A* sehingga berpengaruh pada waktu eksekusi algoritma A*

SOURCE CODE IMPLEMENTASI DAN VIDEO YOUTUBE

Berikut adalah source code yang digunakan untuk implementasi.

https://github.com/AryoBama/implementasi_makalah_stima_2025

Berikut adalah pranala video youtube

https://youtu.be/rTFM8JKwDwQ (apabila pranala youtube rusak, akan dipindahkan ke readme pada repository github)

UCAPAN TERIMA KASIH

Puji syukur dipanjatkan kepada Tuhan Yang Maha Esa yang telah melimpahkan rahmat dan karuniaNya sehingga penyusunan makalah ini dapat diselesaikan dengan lancar. Penulis ingin menyampaikan rasa terima kasih kepada keluarga serta teman – teman karena atas dukungan mereka selama proses pengerjaan makalah sehingga makalah dapat diselesaikan dengan baik. Ucapan terima kasih juga diucapkan kepada para pengajar mata kuliah IF2211 Strategi Algoritma, khususnya Dr. Ir. Rinaldi, M.T. selaku dosen pengampu kelas 02 yang telah memberikan ilmu yang bermanfaat dalam penyusunan makalah ini. Harapan penulis, makalah ini dapat menjadi sumber referensi yang berguna, baik bagi para pembelajar yang tertarik dengan bidang ilmu ini maupun sebagai acuan bagi penulis sendiri di masa

REFERENCES

[1] Orange Void, Sokoban Original & Extra (diakses 24 Juni 2025)

[2] R. Munir, "Penentuan rute (Route/Path Planning) - Bagian 1," https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/17-Algoritma-Branch-and-Bound-(2025)-Bagian1.pdf (diakses 23 Juni 2025).

[3] R. Munir, "Penentuan rute (Route/Path Planning) - Bagian 2," https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf (diakses 23 Juni 2025).

[4] A. Bama, "Penerapan Algoritma Pencarian A* dalam Sistem Pathfinding Mob pada Game Minecraft," https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/Makalah/Makalah-IF1220-Matdis-2024%20(132).pdf (diakses 23 Juni 2025).

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, 24 Juni 2025

Aryo Bama Wiratama, 13523088