TESIS – IF 185401

MULTI-GOAL PROCEDURAL CONTENT GENERATION MENGGUNAKAN DEEP REINFORCEMENT LEARNING

EVAN KUSUMA SUSANTO
05111950010013

Dosen Pembimbing
Prof. Ir. Handayani Tjandrasa, M.Sc, Ph.D

Departemen Teknik Informatika
Fakultas Teknologi Elektro dan Informatika Cerdas
Institut Teknologi Sepuluh Nopember
2022

LEMBAR PENGESAHAN TESIS

Tesis disusun untuk memenuhi salah satu syarat memperoleh gelar
Magister Komputer (M. Kom.)
di
Institut Teknologi Sepuluh Nopember
Oleh:

EVAN KUSUMA SUSANTO
NRP: 05111950010013

Tanggal Ujian: 14 Januari 2022
Periode Wisuda: Maret 2022

Disetujui oleh:
Pembimbing:

1. Prof. Ir. Handayani Tjandrasa, M.Sc, Ph.D .....
   NIP: 194908231976032001

Penguji:

1. Dr. Ahmad Saikhu, S,Si, MT. .....
   NIP: 197107182006041001

2. Dr. Eng. Nanik Suciati, S.Kom, M.Kom. .....
   NIP: 195908171987021002

3. Dr. Darlis Herumurti, S.Kom., M.Kom. .....
   NIP: 195908171987021002

Kepala Departemen Teknik Informatika
Fakultas Teknologi Elektro dan Informatika Cerdas

Dr. Eng. Chastine Fatichah, S.Kom, M.Kom
NIP. 197512202001122002

MULTI-GOAL PROCEDURAL CONTENT GENERATION MENGGUNAKAN DEEP REINFORCEMENT LEARNING

Nama mahasiswa	:	Evan Kusuma Susanto
NRP	:	05111950010013
Pembimbing	:	Prof. Ir. Handayani Tjandrasa, M.Sc, Ph.D

ABSTRAK

Mendesain sebuah *level* pada *game* adalah sebuah pekerjaan yang memerlukan waktu cukup lama dan sulit diotomasikan. Hal ini karena pada proses desain diperlukan pengalaman dan juga diperlukan pemahaman yang cukup mendalam terhadap mekanisme dan cara kerja dari *game* yang akan didesain. Telah banyak dilakukan penelitian untuk *procedural content generation* menggunakan metode *search based* dan metode *machine learning*. Namun semua metode yang ditemukan masih memerlukan metode evaluasi yang memerlukan ahli, perlu *data training* dalam jumlah besar dari ahli, atau hasilnya tidak dapat dikustomisasi sesuai permintaan.

Penelitian ini bertujuan membangun sebuah metode *procedural level generation* yang dapat membuat sebuah *level* sesuai dengan keinginan pengguna. *Level generator* yang dibuat dilatih dengan menggunakan metode *multi-goal deep reinforcement learning*. Untuk dapat melakukan hal ini, proses mendesain sebuah *level game* dibuat menjadi sebuah proses iteratif. *Level generator* hanya dapat memodifikasi satu bagian dari level kemudian sebuah *environment* mengevaluasi kriteria apa saja yang berhasil dipenuhi. *Level generator* kemudian akan mengulangi proses modifikasi hingga semua kriteria yang diminta pengguna berhasil terpenuhi. *Reinforcement learning* digunakan agar *level generator* dapat mempelajari sendiri modifikasi apa saja yang perlu dilakukan agar level memenuhi semua kriteria yang diminta tanpa perlu campur tangan ahli.

Pada penelitian ini dibuat sebuah *level generator* untuk *game* Zelda versi GVGAI. Digunakan 3 ukuran *level* yang berbeda, yaitu 6×6, 7×9, dan 7×11. *Level generator* dibuat dengan cara melatih sebuah *neural network* dengan algoritma *multi-goal reinforcement learning* DQN + HER dan GCSL. Dari hasil penelitian ditemukan bahwa untuk *game* Zelda, DQN kurang stabil (*success rate* turun dan tidak dapat kembali naik) sehingga tidak dapat membuat *level generator* dengan *success rate* yang tinggi. Metode GCSL mampu membuat *level generator* dengan rata-rata *success rate* 0.838 untuk *level* berukuran 6×6, 0.477 untuk *level* berukuran 7×9, 0.364 untuk *level* berukuran 7×11.

Kata kunci: *procedural content generation, level design, deep reinforcement learning, hindsight experience replay*.

MULTI-GOAL PROCEDURAL CONTENT GENERATION MENGGUNAKAN DEEP REINFORCEMENT LEARNING

Student Name	:	Evan Kusuma Susanto
Student Number	:	05111950010013
Mentor	:	Prof. Ir. Handayani Tjandrasa, M.Sc, Ph.D

ABSTRACT

Designing a video game level takes a long time. It is challenging to automate because the design process requires experience and a deep understanding of the mechanics and workings of the game. Many attempts have been done for procedural content generation using search-based and machine learning methods. However, all the previous methods still require evaluation methods made by experts, require large amounts of training data from experts, or the results cannot be customized according to the user's demand.

This study aims to build a procedural level generation method that can create a level based on user criteria. The level generator is trained using the multi-goal deep reinforcement learning method. To do this, game level design process is transformed into an iterative process. The level generator can only modify one part of the level then an environment evaluates what criteria have been met. The level generator then iteratively modifies the level until every user criteria have been met. Reinforcement learning is used so that the level generator can learn what modifications need to be done so that the level meets all the criteria requested without the need for expert intervention.

In this research, a level generator is made for the GVGAI version of the Zelda game. Three different level sizes are used, which are 6×6, 7×9, an7×11. The generator level is created by training a neural network with the multi-goal reinforcement learning algorithm DQN + HER and GCSL. This study found that DQN is less stable (success rate goes down and cannot go back up), so it cannot make a generator level with a high success rate. The GCSL method can create a generator level with an average success rate of 0.838 for a 6×6 level, 0.477 for a 7×9 level, 0.364 for an 7×11 level.

Keywords: *procedural content generation*, *level design*, *deep reinforcement learning*, *hindsight experience replay*.

KATA PENGANTAR

Puji syukur penulis panjatkan ke hadirat Tuhan Yang Maha Esa, karena berkat, rahmat dan anugerah-Nya yang begitu besar, penyusunan penelitian berjudul "Multi-Goal Procedural Content Generation Menggunakan Deep Reinforcement Learning" ini dapat diselesaikan dengan baik.

Dalam pengerjaan penelitian ini, penulis telah mendapatkan banyak bantuan, motivasi, dan bimbingan dari berbagai pihak. Oleh karena itu, pada kesempatan ini, penulis ingin mengucapkan terima kasih kepada:

1. Tuhan Yang Maha Esa atas berkat, rahmat, ilmu, serta berbagai pertolongan yang diberikan sehingga penulis dapat menyelesaikan penelitian ini dengan baik.
2. Orang tua dan keluarga yang telah memberi dukungan dan motivasi kepada penulis baik secara material maupun spiritual dalam pengerjaan penelitian ini.
3. Prof. Ir. Handayani Tjandrasa, M.Sc, Ph.D. selaku dosen pembimbing yang telah membimbing dan mengarahkan dalam pengerjaan penelitian ini.
4. Seluruh dosen, staf tata usaha dan karyawan perpustakaan Jurusan Teknik Informatika, Institut Teknologi Sepuluh Nopember yang telah memfasilitasi penulis sejak awal kuliah.
5. Seluruh keluarga, rekan-rekan, dan pihak-pihak lain yang tidak dapat disebutkan satu persatu atas dukungan dan bantuan yang telah diberikan kepada penulis.

Penulis menyadari bahwa masih banyak kekurangan pada laporan penelitian ini. Oleh karena itu, kritik dan saran yang membangun dari berbagai pihak sangat diharapkan dalam penyempurnaan penelitian ini. Akhir kata, semoga penelitian ini memberikan inspirasi dan manfaat bagi pembacanya.

Surabaya, Januari 2022

[*Halaman ini sengaja dikosongkan*]

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

DAFTAR ALGORITMA

BAB 1
PENDAHULUAN

Bab ini merupakan penjelasan mengenai beberapa hal dasar dalam pembuatan penelitian. Hal-hal yang dimaksud meliputi latar belakang, perumusan masalah, tujuan penelitian, manfaat penelitian, kontribusi penelitian, serta batasan masalah.

1. 1. Latar Belakang

Istilah "*level*" pada sebuah *game* didefinisikan sebagai daerah yang dapat dieksplorasi oleh pemain untuk menyelesaikan sebuah tujuan. *Level* dari sebuah *game* bisa memiliki berbagai macam bentuk. Misalnya untuk *game* petualangan, *level* dapat berupa peta sebuah daerah yang hendak dieksplorasi. Untuk sebuah game *puzzle* (misalnya sudoku), sebuah *level* adalah kombinasi angka dan petak kosong yang harus diisi dengan angka.

Setiap *game* biasanya memiliki beberapa *level* dengan tingkat kesulitan masing-masing. Biasanya kumpulan *level* ini disusun sedemikian sehingga *level* dengan tingkat kesulitan rendah diberikan di awal *game* dan *level* dengan tingkat kesulitan tinggi diberikan di akhir *game*. Pengaturan ini bertujuan agar pemain selalu tertarik untuk memainkan *game*. *Level* yang selalu mudah akan membuat pemain cepat bosan dan *level* yang selalu sulit akan membuat pemain frustrasi dan tidak ingin melanjutkan permainan (Schell, 2014).

Salah satu bagian paling penting dalam membuat sebuah *game* yang menarik adalah proses *level designing*. Sebuah *level* pada *game* harus didesain sedemikian sehingga sebuah *level* menjadi cukup sulit agar permainan menjadi menantang, tetapi tidak terlalu sulit sehingga pemain masih dapat menyelesaikannya. Untuk dapat mendesain sebuah *level* seperti itu, diperlukan pengalaman dalam mendesain *game* dan juga diperlukan pemahaman yang cukup mendalam terhadap mekanisme dan cara kerja dari *game* yang akan didesain. Hal ini menyebabkan *level designing* menjadi sebuah pekerjaan yang memakan waktu cukup lama dan sulit diotomasikan.

Salah satu upaya yang dilakukan dalam melakukan otomasi pembuatan *level* adalah pembuatan metode *procedural content generation*. *Procedural content generation* (Togelius *et al.*, 2011) adalah sebuah proses pembuatan konten secara

acak tetapi mengikuti beberapa prosedur yang telah ditentukan. Hal ini dilakukan dengan cara melakukan parameterisasi beberapa fitur yang ada dalam sebuah konten, kemudian menggunakan prosedur yang telah ditentukan untuk membuat konten berdasar *parameter* yang diberikan.

Procedural content generation juga dapat dilakukan untuk membuat *level* dari sebuah *game*. *Procedural content generation* yang dilakukan untuk membuat *level game* biasa disebut *procedural level generation*. *Procedural level generation* mempercepat proses pembuatan *level game* karena dengan metode ini *game* dapat secara otomatis menyusun *level*-nya sendiri dalam jumlah banyak dan masing-masing *level* memiliki kualitas yang cukup baik. Namun metode ini masih memiliki masalah yang sama dengan cara sebelumnya, yaitu diperlukannya pengalaman dan pemahaman terhadap *game* untuk dapat menyusun prosedur pengacakan yang dapat menghasilkan *level* yang baik.

Perkembangan berikutnya dari *procedural level generation* adalah menggunakan algoritma yang dapat secara otomatis menentukan prosedur pembuatan *level game*. Penelitian sebelumnya (Togelius *and* Shaker, 2016) telah menggunakan berbagai metode, misalnya dengan menggunakan *heuristic search* untuk menentukan *parameter* optimal untuk membuat *level* yang berkualitas. Metode lainnya menggunakan data aksi yang dilakukan pemain dalam *game* untuk mendesain *level* yang menantang. Perkembangan lain adalah menggunakan metode *deep learning* (Summerville *et al.*, 2018) yang mempelajari contoh-contoh *level* yang baik, kemudian berusaha menghasilkan *level* yang mirip tetapi lebih bervariasi dan tetap menantang. Penggunaan metode-metode ini mengurangi beban dari pembuat *level*, tetapi memerlukan data yang cukup banyak untuk dapat menghasilkan *generator level* yang baik.

Inovasi baru yang ditemukan untuk *procedural level generation* adalah penggunaan *reinforcement learning* (Khalifa *et al.*, 2020). Menggunakan *reinforcement learning*, dapat dibuat sebuah kecerdasan buatan yang belajar untuk menyelesaikan permasalahan dari *trial and error*. Proses pembuatan kecerdasan buatan pada umumnya menggunakan metode *heuristic search* atau dengan belajar dari data-data yang sudah pernah tersedia (*supervised learning*). Kecerdasan pada *reinforcement learning* belajar dengan cara mencoba aksi yang dapat dilakukan dan

mempelajari efeknya melalui umpan balik yang diterima. Metode ini biasanya digunakan untuk membuat sebuah kecerdasan buatan yang dapat memainkan sebuah *game* tanpa diberi pengetahuan sebelumnya tentang *game* tersebut.

Penggunaan *reinforcement learning* untuk *procedural level generation* sudah dilakukan sebelumnya pada sebuah penelitian terbaru. Pada penelitian ini, dibuat sebuah kecerdasan buatan yang dapat memodifikasi sebuah *level game* yang diberikan untuk meningkatkan kompleksitas *level* tersebut. Untuk membuat *level* baru, pertama-tama dibuat sebuah *level* secara acak, kemudian *level* diberikan kepada kecerdasan buatan untuk dimodifikasi secara iteratif sampai terbentuk *level* yang dapat diselesaikan dan dengan kompleksitas maksimum. Dengan menggunakan metode *procedural level generation* tidak diperlukan *designer* yang berpengalaman dan data yang banyak, tetapi cukup dengan membuat *game* yang hendak digunakan dan *program* untuk mengevaluasi kompleksitas sebuah *level*.

Kekurangan dari metode sebelumnya adalah kecerdasan buatan yang dihasilkan hanya dioptimasi untuk menghasilkan sebuah *level* yang paling kompleks. Padahal pada dunia nyata, *level* yang diperlukan tidak selalu yang paling kompleks tetapi disesuaikan dengan kemampuan pemain. Selain itu, tidak dapat diatur fitur-fitur apa saja yang akan muncul pada *level*. Misalnya, pada *level* tidak dapat diatur jumlah musuh yang muncul, jenis musuh apa saja yang boleh muncul, jumlah pintu yang ada pada *level*, dan lain sebagainya.

Pada penelitian ini diusulkan modifikasi terhadap algoritma yang lama berupa penambahan *multi-goal reinforcement learning* (Schaul *et al.*, 2015). Pada penelitian sebelumnya digunakan algoritma *Proximal Policy Optimization* (PPO) (Schulman *et al.*, 2017). Algoritma ini terbatas hanya pada satu tujuan yaitu untuk memaksimalkan kompleksitas dari *level* yang dihasilkan. Pada penelitian ini akan digunakan algoritma *Hindsight Experience Replay* (HER) (Andrychowicz *et al.*, 2017), sebuah algoritma *multi-goal deep reinforcement learning*. Penggunaan HER pada penelitian ini dapat membuat *level* yang dihasilkan dapat lebih beragam. Misalnya, dapat ditentukan di awal bahwa *level* yang diminta memiliki tingkat kesulitan rendah, jumlah musuh yang muncul 1, dan musuh yang muncul adalah musuh tipe A. Atau dapat ditentukan juga bahwa *level* yang diminta memiliki

tingkat kesulitan sedang, jumlah musuh yang muncul 3 musuh tipe A dan 2 musuh tipe B.

1. 2. Rumusan Masalah

Rumusan masalah yang diangkat dalam penelitian ini adalah sebagai berikut:

1. Bagaimana cara agar *level* yang dihasilkan dengan metode *deep reinforcement learning* dapat dikustomisasi?
2. Seberapa akurat model yang dihasilkan dalam memenuhi permintaan pengguna dalam pembuatan *level*?

1. 3. Tujuan Penelitian

Tujuan yang akan dicapai dalam penelitian ini adalah membangun metode *procedural level generation* dengan *reinforcement learning* yang dapat dikustomisasi dengan cara menggabungkan *PCGRL* dan algoritma *multi-goal deep reinforcement learning*

1. 4. Manfaat Penelitian

Manfaat dari penelitian ini diantaranya adalah untuk mengembangkan sebuah metode untuk melakukan *procedural level generation* menggunakan *multi-goal deep reinforcement learning*. Hal ini dilakukan dengan cara menggabungkan *PCGRL* dan algoritma *multi-goal deep reinforcement learning*. Metode ini diharapkan dapat membuat sebuah sistem yang dapat menghasilkan *level game* secara otomatis tanpa diperlukannya *dataset* atau pembuat *level* yang berpengalaman untuk menghasilkan *level* dengan spesifikasi yang diinginkan dalam jumlah banyak dalam waktu singkat.

1. 5. Kontribusi Penelitian

Kontribusi yang diharapkan dari penelitian ini adalah dibuatnya metode baru untuk melakukan *procedural level generation* menggunakan *multi-goal deep reinforcement learning*.

1. 6. Batasan Masalah

Batasan masalah pada penelitian ini adalah:

1. *Procedural content generation* pada penelitian ini hanya dibatasi pada pembuatan *level* saja
2. Implementasi dilakukan dengan menggunakan bahasa pemrograman Python dan implementasi arsitektur *deep reinforcement learning* menggunakan *library Stable Baselines*.
3. Penelitian ini akan menggunakan *game* “ Z e l d a ” v e r s i “ G V G A I ” obyek penelitian. Metode evaluasi yang akan digunakan akan dimodifikasi dari penelitian sebelumnya.
4. Komponen *level* yang dapat dikustomisasi hanya dapat memilih dari pilihan kustomisasi yang disediakan, tidak bisa bebas.
5. Ukuran *level* yang akan dibuat ditentukan di awal yaitu sebuah papan berukuran dari 6×6 , 7×9 , dan 7×11.

[*Halaman ini sengaja dikosongkan*]

BAB 2
DASAR TEORI DAN KAJIAN PUSTAKA

Pada bab ini akan dibahas referensi-referensi yang terkait dengan metode yang akan digunakan untuk menyelesaikan permasalahan yang telah dijabarkan pada bagian latar belakang penelitian. Bab ini diawali dengan penjelasan tentang dasar teori dari permasalahan, metode lama dan penelitian yang pernah dilakukan, serta metode baru yang diusulkan pada penelitian ini. Setelah itu, akan dibahas inovasi apa yang akan ditambahkan pada metode baru.

2. 1. *Procedural Content Generation*

Procedural content generation (PCG) (Shaker, Togelius *and* Nelson, 2016) dapat didefinisikan sebagai pembuatan konten sebuah *game* dengan *input* yang terbatas atau tidak langsung dari pengguna. Sebuah perangkat lunak dianggap melakukan PCG apabila perangkat tersebut dapat membuat konten *game* sendiri atau dengan bantuan pemain atau desainer. PCG dalam konteks *game* dapat merujuk pada pembuatan *level*, desain obyek yang ada dalam *game*, tekstur, dan lain sebagainya. Selain itu, penelitian yang berhubungan dengan PCG dalam *game* biasanya terbagi menjadi 2 jenis, yaitu PCG menggunakan metode *search based* dan metode *machine learning*.

2.1.1 Metode *search based*

Sebuah metode PCG dianggap sebagai *search based* apabila pada pembuatan konten digunakan algoritma optimasi, *search* atau evolutionary. Pada metode ini, algoritma *search* dan optimasi digunakan untuk mencari *level* yang paling optimal sesuai dengan batasan-batasan yang telah diberikan. Pencarian *level* ini biasanya dilakukan dengan cara melakukan perubahan kecil pada *level* secara terus menerus pada *level* yang diberikan. Dalam proses pencarian ini, perubahan yang memberi efek positif pada *level* akan disimpan dan perubahan yang merusak *level* akan dibuang. Dengan melakukan modifikasi secara terus menerus, diharapkan *level*

yang dihasilkan di akhir proses ini akan menjadi optimal. Pada umumnya, sebuah metode PCG *search based* memerlukan 3 komponen utama ini:

1. Algoritma *search*. Pada PCG, algoritma *search* adalah komponen yang akan menentukan modifikasi apa saja yang akan dilakukan untuk mengarahkan *level* saat ke arah yang lebih baik.

2. Representasi konten. Agar dapat dimodifikasi oleh algoritma *search*, *level* harus direpresentasikan dalam *program*. Representasi *level* ini dapat berupa berbagai macam struktur data, misalnya *array*, *graph*, *string*, atau sebuah *object*. Pemilihan representasi *level* cukup penting karena representasi *level* dapat mempengaruhi apa saja yang dapat dimodifikasi oleh algoritma *search*.

3. *Evaluation function*. *Evaluation function* adalah komponen yang menentukan kualitas dari *level* saat ini. Fungsi inilah yang digunakan untuk menentukan apakah modifikasi yang dilakukan oleh algoritma *search* memberi efek positif atau negatif dan menentukan arah dari modifikasi yang dilakukan oleh algoritma *search*. Dalam sebuah PCG bisa terdapat lebih dari 1 buah *evaluation function*.

2.1.2 Metode *machine learning*

Perkembangkan *machine learning* yang pesat membuat munculnya inovasi baru yaitu penggunaan *machine learning* sebagai salah satu alternatif dalam PCG (Liu *et al.*, 2020). *Procedural content generation* mengunakan *machine learning* dilakukan dengan cara melatih sebuah model dengan menggunakan data *level* yang sudah dibuat sebelumnya. Model yang sudah dilatih tersebut kemudian dapat menghasilkan sebuah *level* yang mirip dengan *level* yang pernah dipelajari. Contoh dari penggunaan *machine learning* dalam PCG adalah penggunaan Generative Adversarial *Network* untuk membuat sebuah *level* pada *game* "*Super Mario Bros*" (Volz *et al.*, 2018) .

Penggunaan *machine learning* untuk PCG memiliki kelebihan dan kekurangan dibandingkan dengan metode *search based*. Pada metode *search based*, diperlukan desainer yang dapat memahami cara kerja *game* secara mendalam dan

dapat mengatur *parameter search*, representasi *level*, dan *evaluation function*. Sedangkan ada metode maching *learning*, dapat dilatih sebuah model yang dapat belajar dengan sendirinya cara kerja *game* dan modifikasi apa saja yang dapat membuat *level* menjadi lebih baik. Kelemahan utama dari metode *machine learning* biasa adalah diperlukannya banyak data yang digunakan untuk proses pelatihan model. Data tersebut harus dibuat sebagus mungkin oleh desainer yang ahli di bidangnya, sehingga metode *machine learning* biasa belum bisa menghilangkan ketergantungan terhadap ahli.

2. 2. *Reinforcement learning*

Reinforcement learning (Sutton *and* Barto, 1998) adalah kategori *machine learning* di mana komputer tidak diberi "supervisi" atau diberi panduan oleh *user* secara langsung. Pada *reinforcement learning*, komputer akan dihadapkan dengan sebuah *environment* dan belajar dari hasil interaksi antara komputer dengan *environment* tersebut. *User* tidak memberi panduan berupa jawaban yang diharapkan dari komputer secara langsung, tetapi *user* membiarkan komputer berinteraksi dengan *environment* dan akan memberi *reward* (imbalan) kepada komputer ketika komputer berhasil melakukan sesuatu hal yang baik dan memberi *punishment* (hukuman) ketika komputer melakukan suatu hal yang buruk.

Reinforcement learning biasa dilakukan ketika permasalahan yang hendak diselesaikan cukup rumit untuk diformulasikan secara langsung tetapi dapat ditentukan dengan mudah apakah metode penyelesaian yang dilakukan baik atau buruk. Contoh sederhana yang memanfaatkan *reinforcement learning* adalah menerbangkan helikopter secara otomatis. Manuver yang harus dilakukan pada suatu keadaan tidak mudah ditentukan, tetapi ketika sebuah manuver dilakukan dan helikopter jatuh, maka dapat dipastikan dengan mudah bahwa manuver tadi adalah sebuah tindakan yang salah untuk kondisi tersebut.

2. 3. *Markov Decision Process*

Markov Decision Process (MDP) (Bellman, 1957) merupakan sebuah formalisasi dari *sequential decision making* (pengambilan keputusan berurutan), di

mana keputusan yang diambil akan mempengaruhi situasi dan *reward* berikutnya. Dalam MDP dibentuk sebuah model matematika ideal dari sebuah permasalahan *reinforcement learning* sehingga dapat dibuat pernyataan teoritis terhadap permasalahan terebut. Dengan MDP dapat dimodelkan pengambilan keputusan di mana hasil akhir yang terjadi sebagian dipengaruhi oleh keputusan yang diambil dan sebagian dipengaruhi oleh peluang. Dalam MDP terdapat beberapa komponen penting yang akan digunakan dalam *reinforcement learning*. Berikut adalah komponen-komponen tersebut:

1. *State*: *state* adalah kumpulan dari keadaan yang dapat terjadi pada suatu *environment*. Selain itu, *state* bisa disebut juga sebagai segala sesuatu yang menggambarkan keadaan kondisi *environment* saat itu. *State* biasanya adalah hal-hal yang berubah dari waktu ke waktu dari suatu sistem. Pada MDP, sebuah *state* disimbolkan dengan simbol s, sedangkan himpunan dari seluruh *state* yang ada pada MDP tersebut disimbolkan dengan simbol $\mathcal{S}$.

2. *Transition*: *transition* menggambarkan aturan-aturan yang terdapat pada *environment* tersebut. *Transition* juga menentukan apa yang akan terjadi apabila sebuah aksi diambil ketika berada di sebuah *state*. *Transition* digambarkan sebagai sebuah fungsi dengan 3 *parameter*, yaitu *state* awal, aksi yang diambil pada *state* tersebut, dan *state* akhir yang hendak dituju, dan mengembalikan sebuah nilai berupa peluang terjadinya perubahan dari *state* awal ke *state* tujuan apabila aksi dilakukan pada *state* awal. *Transition* disimbolkan dengan huruf T atau P.

3. *Action*: *action* adalah segala sesuatu yang dapat dilakukan pada suatu *state*, dan melakukan sebuah *action* dapat menyebabkan terjadinya perubahan *state* melalui *transition*. Sebuah aksi disimbolkan dengan huruf a dan himpunan aksi yang dapat dilakukan pada suatu *state* s disimbolkan dengan $\mathcal{A}(\mathrm{s})$

4. *Reward*: *Reward* adalah suatu nilai skalar yang diberikan ketika suatu *state* dicapai. *Reward* dapat berupa nilai positif untuk menandakan *state* yang baik dan dapat bernilai negatif untuk menandakan *state* yang buruk. *Reward* inilah yang menjadi umpan balik untuk komputer yang membantu menentukan apakah suatu tindakan yang diambil baik atau buruk. *Reward* dapat disimbolkan dengan huruf r

atau R(s) untuk menunjukkan *reward* yang diterima apabila *state* s dicapai.

5. *Policy*: *policy* dapat digambarkan sebagai sebuah fungsi yang menerima sebuah *state* sebagai *parameter* dan mengembalikan sebuah *action* sebagai nilai kembalian. *Policy* disimbolkan dengan π dan π(s) mengembalikan sebuah aksi a yang akan diambil pada *state* s ketika *policy* π diikuti. Tujuan akhir dari MDP adalah untuk menentukan *optimum policy*, yaitu *policy* yang menghasilkan total *reward* terbanyak yang dapat dicapai.

2. 4. Persamaan Bellman

Untuk menentukan apakah suatu keputusan merupakan keputusan yang baik atau buruk secara obyektif, diperlukan sebuah pengukuran yang didefinisikan secara jelas serta telah dirancang untuk memenuhi tujuan yang hendak dicapai. *Bellman equation* (Bellman, 1957) salah satu persamaan yang dapat digunakan untuk mengukur seberapa baik sebuah keputusan yang diambil pada suatu kondisi tertentu dengan melibatkan keuntungan yang dapat diperoleh di masa depan sebagai konsekuensi dari pengambilan keputusan tersebut. Rumus Persamaan Bellman (Bellman, 1957) dapat dilihat di persamaan 2.1:

$$V(s) = \max_{a}(R(s,a) + \gamma \Sigma_{s'} T(s,a,s')V(s')) \quad (2.1)$$

Simbol $V(s)$ menyatakan *value function*. *Value function* adalah sebuah fungsi yang menyatakan *reward* potensial maksimal yang dapat diperoleh jika *state* tersebut berhasil dicapai. Keputusan yang baik untuk diambil pada suatu *state* adalah keputusan yang dapat mengarahkan kondisi *state* sekarang menjadi kondisi *state* dengan *value* yang terbaik yang dapat dicapai. Simbol $R(s,a)$ dan $T(s,a,s')$ menyatakan *reward* dan *transition* pada MDP. Seperti dijelaskan di atas, *reward* menyatakan nilai skalar yang diterima apabila pada suatu *state* s diambil aksi a. Sedangkan *transition* menyatakan peluang terjadinya perubahan *state* dari *state* s menjadi *state* s′ ketika aksi a dilakukan pada saat berada pada *state* s.

Ekspresi $\max_a(R(s,a) + \gamma\Sigma_{s'}T(s,a,s')V(s'))$ adalah nilai maksimum dari ekspektasi *value* yang dapat diperoleh dari *state-state* yang dapat dicapai dari *state* asal akibat aksi yang diambil. Ekspresi ini dievaluasi dengan cara dikalkulasi nilai ekspektasi dari *value function* yang dapat diperoleh ketika sebuah aksi diambil. Kemudian diambil nilai ekspektasi paling besar diantara semua nilai yang dihasilkan tiap aksi.

Persamaan Bellman juga memiliki bentuk lain. Pada persamaan pertama dihitung nilai ekspektasi *reward* maksimum yang dapat dicapai ketika mencapai sebuah *state*. Bentuk kedua dari persamaan Bellman mengkalkulasi nilai ekspektasi *reward* maksimum yang dapat dicapai ketika melakukan sebuah aksi pada sebuah *state*. Persamaan Bellman (Bellman, 1957) yang kedua dirumuskan pada persamaan 2.2:

$$Q(s,a) = R(s,a) + \gamma\Sigma_{s'}T(s,a,s') \max_{a'} Q(s',a') \quad (2.2)$$

Simbol $Q(s,a)$ menyatakan *Q function*. *Q function* adalah sebuah fungsi yang menyatakan *reward* potensial maksimal yang dapat diperoleh ketika sebuah aksi dieksekusi pada sebuah *state*. Sama seperti persamaan sebelumnya, diperlukan nilai dari $R(s,a)$ dan $T(s,a,s')$ untuk pasangan *state* dan aksi yang dilakukan. Ekspresi $\max_{a'} Q(s',a')$ adalah nilai *Q function* maksimum yang dapat dicapai pada *state* berikutnya.

2. 5. *Q learning*

Q learning (Watkins *and* Dayan, 1992) adalah salah satu algoritma *reinforcement learning value based* yang memanfaatkan *Q function* untuk memperoleh *policy* optimal. *Q function* adalah sebuah fungsi yang menyatakan *reward* potensial maksimal yang dapat diperoleh ketika sebuah aksi dieksekusi pada sebuah *state*. *Q function* disimbolkan sebagai Q(s,a) yang menerima *parameter* berupa *state* saat itu dan aksi yang diambil. Secara intuisi, nilai dari *Q function* adalah nilai skor tertinggi yang dapat diperoleh apabila pada *state* s diambil aksi a. Nilai *Q function* dari sebuah aksi pada sebuah *state* dapat diperoleh dari

reward yang diterima setelah aksi dilakukan ditambah *Q function* maksimum dari *state* berikutnya. Dari intuisi di atas dapat dirumuskan persamaan *Q function* (Watkins *and* Dayan, 1992) sebagai persamaan rekursif 2.3:

$$Q(s,a) = r + \gamma \max_{a'} Q(s',a') \tag{2.3}$$

Secara definisi, skor tertinggi pada sebuah *state* dapat diperoleh apabila aksi dengan *Q function* tertinggi diambil. Oleh karena itu, *optimum policy* adalah *policy* yang selalu mengambil aksi dengan *Q function* tertinggi untuk masing-masing *state*. *Policy* untuk Q-*learning* dirumuskan sebagai persamaan 2.4 (Watkins *and* Dayan, 1992):

$$\pi(s) = argmax_a \, Q(s,a) \tag{2.4}$$

Pada saat inisialisasi, nilai *Q function* dari masing-masing pasang *state* dan *action* diisi dengan nilai *random*. Setiap kali aksi telah diambil dan *reward* telah diperoleh, akan dilakukan *update* pada nilai dari *Q function* untuk pasangan *state* dan aksi tersebut. Persamaan 2.5 adalah rumus yang digunakan untuk *update* nilai *Q function* (Watkins *and* Dayan, 1992):

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R + \gamma \max_{a'} Q(s',a') - Q(s,a)) \tag{2.5}$$

Di mana nilai $Q(s,a)$ adalah *Q function* untuk aksi yang baru saja diambil, nilai α adalah *learning rate*, γ adalah *discount factor*, dan $\max_{a'} Q(s',a')$ adalah nilai maksimum *Q function* yang mungkin pada *state* berikutnya. Sama seperti metode *model free* lainnya, nilai awal dari estimasi *Q function* adalah *random* dan diperlukan percobaan berulang kali agar nilai estimasi dapat diperbaiki. Selama metode *Q learning* diikuti dan percobaan dilakukan dalam jumlah banyak, maka isi dari *Q function* akan menjadi benar.

2. 6. *Deep* Q *Network* (DQN)

Deep Q Network (Mnih *et al.*, 2013) adalah sebuah arsitektur *deep reinforcement learning* yang dikembangkan oleh *Deep*Mind pada tahun 2013. Inovasi yang dilakukan pada DQN adalah penggunaan *convolutional neural network* sebagai *function approximator* dan pelatihan sebuah *neural network* menggunakan metode *Q learning*. Hal ini membuat jumlah pasangan *state* dan *action* yang banyak menjadi berkurang drastis karena *state* yang mirip dapat dipangkas menjadi satu menggunakan *convolutional neural network*.

```
Inisialisasi replay memory D dengan kapasitas N
Inisialisasi weight Q network (θ) dengan nilai random
Inisialisasi target network  Q̂ dengan nilai weight θ⁻ = θ
For episode = 1,M do
    Terima state awal s dari environment
    For t = 1,T do
        Dengan peluang ϵ pilih aksi random a_t
        Jika tidak a_t = argmax_a Q(s_t, a)
        Lakukan aksi a_t dan terima reward r_t dan state s_{t+1}
        Simpan transisi (s_t, a_t, r_t, s_{t+1}) di D
        Sampel random minibatch transisi (s_j, a_j, r_j, s_{j+1})
        jika episode berakhir pada step j + 1 maka
                y_j ← r_j
        jika tidak maka
                y_j ← r_j + γ max_{a'} Q̂(s_{j+1}, a')
        lakukan gradient descent dengan loss (y_j − Q(s_j, a_j))²
    terhadap parameter θ
        Setiap C step sekali,  Q̂ ← Q
    End For
End For
```

Algoritma 2.1 *Deep* Q *Network with Experience Replay* (Mnih *et al.*, 2013)

Metode pelatihan DQN agak berbeda dari *convolutional neural network* biasa. Pada DQN, *dataset* untuk *training* tidak disediakan secara langsung tetapi harus diperoleh dari hasil berinteraksi dengan *environment*. Pada tiap *timestep* t, DQN akan menerima *input* berupa *state* s_t, melakukan aksi a_t, kemudian menerima *state* s_{t+1} dan *reward* r_t akibat melakukan aksi tersebut. Data transisi (s_t, a_t, r_t, s_{t+1}) ini kemudian disimpan dalam sebuah *memory* yang disebut dengan *replay memory*. Secara periodik, *neural network* akan mengambil sekumpulan data

transisi dari *replay memory* sebagai data *training* untuk melakukan *update* bobot menggunakan metode *backpropagation*.

Output dari DQN bukan berupa *one-hot vector* seperti masalah klasifikasi biasa, tetapi estimasi *Q function* dari tiap aksi apabila dilakukan pada *state* saat ini (menjadi regresi nilai *Q function*). Nilai *Q function* target diperoleh dari kalkulasi *Q function* menggunakan persamaan *Q learning* di atas. Dengan adanya estimasi *Q function* untuk masing-masing aksi, maka dapat digunakan *greedy policy* untuk penentuan aksi yang akan diambil (ambil aksi dengan nilai *Q function* yang paling tinggi). Algoritma DQN dapat dilihat pada algoritma 2.1.

2. 7. *Policy Gradient*

Policy gradient (Sutton *et al.*, 1999) adalah salah satu metode *reinforcement learning* yang bersifat *policy based learning*. Pada *policy based learning policy* dicari dan ditentukan secara langsung tanpa memerlukan *value function*. *Policy* akan dikalkulasi dan dimodifikasi sesuai dengan performa agen (total skor yang diterima) ketika berinteraksi dengan *environment* dengan *policy* yang diberikan. Dalam menggunakan *policy based learning*, *policy* didefinisikan sebagai sebuah fungsi yang menerima *input* berupa *state* dan *parameter* berupa vektor *weight* untuk menghasilkan *output* berupa distribusi probabilitas sebuah aksi diambil (*parameterized policy*). Selain itu, diperlukan sebuah *function approximator* untuk mengubah *state* menjadi sebuah representasi yang dapat digunakan dalam operasi matematika.

Metode *policy gradient* adalah salah satu teknik *reinforcement learning* di mana dilakukan optimasi terhadap *parameterized policy* terhadap ekspektasi *reward* dengan menggunakan gradien. Diberikan sebuah *policy* yang menggunakan *parameter* θ dan *cost function* $J(\theta)$. *Cost function* menggambarkan ekspektasi *reward* yang dapat diperoleh apabila *policy* dengan *parameter* θ diikuti. Dicari sebuah kombinasi θ yang dapat memaksimalkan nilai dari $J(\theta)$. Hal ini dilakukan dengan cara dicari *local maximum* dari persamaan $J(\theta)$ dengan cara nilai *parameter* θ digeser searah dengan gradien *policy* terhadap *parameter*. Hal ini dirumuskan pada persamaan 2.6 dan 2.7 (Sutton *et al.*, 1999):

$$\theta_{t+1} = \theta_t + \alpha \nabla_{\theta_t} J(\theta_t) \tag{2.6}$$

$$\nabla_\theta J(\theta) = \begin{pmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{pmatrix} \tag{2.7}$$

Di mana α adalah *parameter step size* yang mengatur seberapa besar langkah yang diambil setiap kali dilakukan *update weight*. Simbol $\nabla_\theta J(\theta)$ disebut sebagai *policy gradient*. *Policy gradient* diperoleh dari turunan parsial *cost function* terhadap masing-masing *parameter weight*. Setelah diperoleh nilai dari *policy gradient*, nilai dari *parameter* bobot yang baru diperoleh dari penjumlahan *parameter* bobot dengan *policy gradient*. Hal ini dilakukan karena *policy gradient* mengarah pada nilai *local maximum* dari *cost function*. Dalam kasus *reinforcement learning*, nilai *cost function* ini dapat diganti menjadi ekspektasi *reward* yang akan diterima jika *policy* π_θ diikuti. Nilai ekspektasi *reward* yang akan diterima dapat dirumuskan dengan persamaan 2.8 (Sutton *et al.*, 1999):

$$\mathbb{E}_{\pi_\theta}[r] = \Sigma_{a \in A}\, \pi_\theta(s, a) R_{s,a} \tag{2.8}$$

2. 8. *Proximal Policy Optimization* (PPO)

Proximal Policy Optimization (PPO) (Schulman *et al.*, 2017) adalah algoritma *reinforcement learning* yang termasuk *policy gradient* dan bersifat *on-policy*. Metode ini merupakan metode terbaru dan terbaik saat ini untuk kategori algoritma *on-policy* metode *policy gradient* dan metode ini digunakan pada penelitian sebelumnya. PPO mengatasi permasalahan yang sering terjadi pada algoritma *policy gradient* di mana *update* yang dilakukan pada *parameter* terlalu besar sehingga mengakibatkan kerusakan pada *policy*. Inovasi yang dilakukan adalah pembatasan nilai *update* yang dilakukan sehingga *policy* yang dihasilkan setelah *update* tidak berbeda jauh dengan *policy* sebelum *update*. Pembatasan ini dirumuskan dengan persamaan 2.9 dan 2.10 (Schulman *et al.*, 2017):

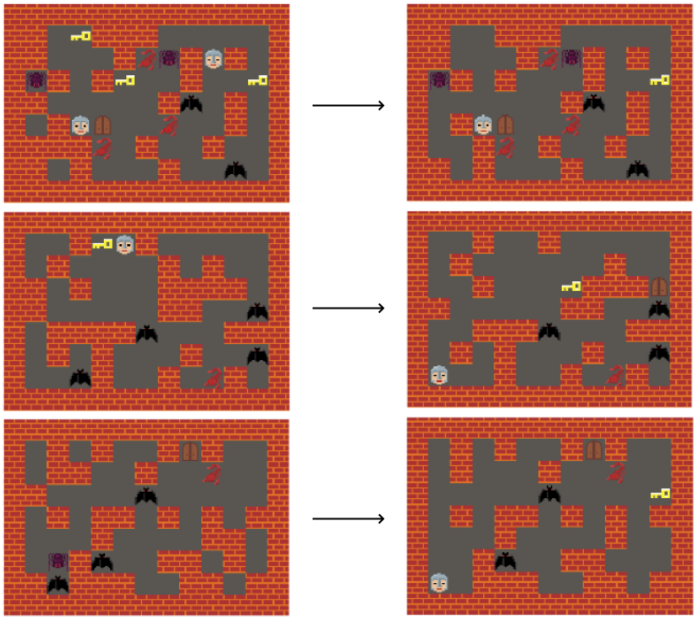
$$\mathrm{L}\,(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a), \mathrm{g}\big(\epsilon, A^{\pi_{\theta_k}}(s,a)\big)\right) \quad (2.9)$$

$$\mathrm{g}(\epsilon,A) = \begin{cases} (1+\epsilon)A & A\geq 0 \\ (1-\epsilon)A & A<0 \end{cases} \quad (2.10)$$

Pada penelitian ini metode PPO tidak digunakan karena PPO hanya dapat mengoptimasi agen untuk memenuhi 1 target yang tidak berubah (pada penelitian sebelumya hal ini adalah *reward* yang diperoleh), padahal pada penelitian ini terdapat lebih dari 1 target dan target yang diminta berubah-ubah di setiap iterasi.

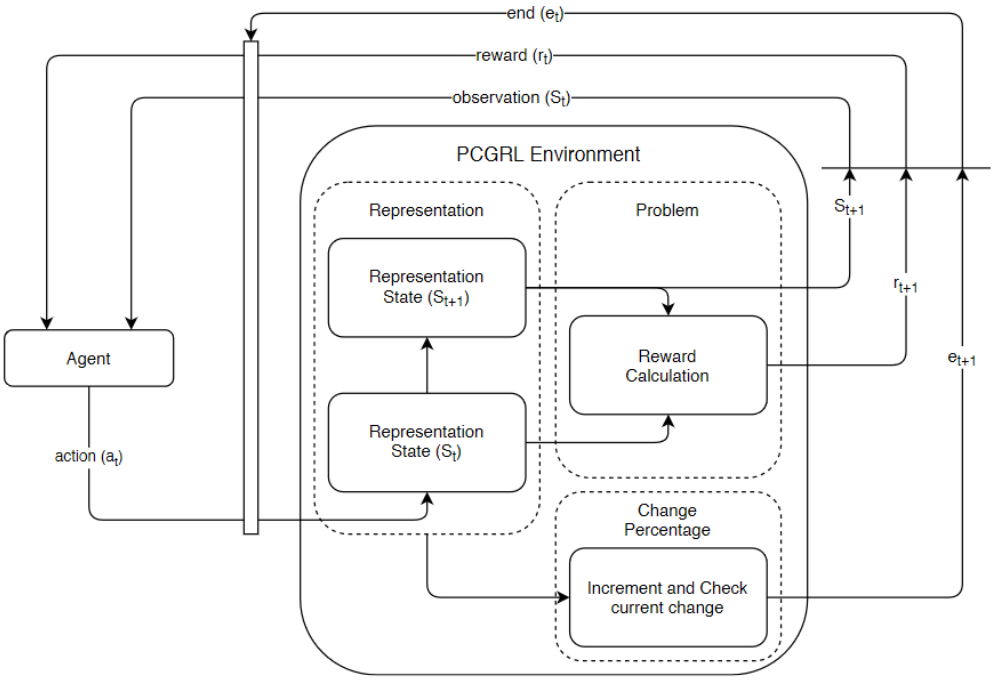
2. 9. *Procedural Content Generation with Reinforcement Learning* (PCGRL)

PCGRL (Khalifa *et al.*, 2020) adalah salah satu inovasi terbaru dalam bidang PCG, di mana pada penelitian ini digunakan algoritma *reinforcement learning* untuk melakukan *procedural level generation*. Keuntungan utama dari penggunaan *reinforcement learning* adalah tidak diperlukannya ahli dan data *training* untuk dapat melatih agen yang dapat menghasilkan *level*. Agen dapat mempelajari sendiri apa yang membuat sebuah *level* menjadi baik dengan cara *trial and error* menggunakan sebuah *environment* yang memberi *reward* positif apabila *level* yang dihasilkan lebih kompleks dari *level* sebelumnya. Contoh hasil kerja dari PCGRL dapat dilihat pada Gambar 2.1



Gambar 2.1. Contoh *Level* yang Didesain oleh PCGRL (Khalifa *et al.*, 2020)

Pada PCGRL, proses pembuatan sebuah *level game* diubah menjadi sebuah permasalahan iteratif. Arsitektur dari sistem ini dapat dilihat pada Gambar 2.2. Agen diberi *input* berupa sebuah *level* (s_t), kemudian agen dilatih dengan menggunakan algoritma *reinforcement learning* PPO untuk dapat melakukan modifikasi yang dapat membuat *level* tersebut menjadi lebih baik (s_{t+1}). Setiap kali agen melakukan modifikasi, *environment* akan melakukan kalkulasi kualitas atau kompleksitas *level* yang baru dibandingkan *level* yang lama. *Environment* akan memberi *reward* (r_t) positif apabila *level* hasil modifikasi lebih kompleks dibandingkan *level* sebelum modifikasi dan *reward* negatif apabila *level* menjadi kurang kompleks atau tidak bisa dimainkan.



Gambar 2.2. Arsitektur Sistem PCGRL (Khalifa *et al.*, 2020)

Kekurangan dari metode PCGRL ini adalah agen yang dihasilkan hanya dioptimasi untuk menghasilkan *level* yang paling kompleks dan paling sulit. Padahal pada praktiknya, tidak semua *level* harus dirancang sesulit mungkin. Selain itu, pada metode ini tidak dapat dilakukan kustomisasi terhadap konten apa saja

yang muncul pada *level*. Misalnya, hampir semua *game* mengeluarkan jenis musuh yang paling mudah di *level* awal dan seiring berjalannya waktu, musuh yang muncul akan semakin kuat dan sulit dikalahkan. Pada metode lama, jenis musuh yang dimunculkan pada *level* tidak bisa dibatasi, sehingga pada setiap *level*, setiap jenis musuh memiliki peluang yang sama untuk muncul. Pada penelitian ini akan dikembangkan sebuah metode yang dapat mengatasi hal ini.

2. 10. *Universal Value Function Approximator*

Universal Value Function Approximator (UVFA) (Schaul *et al.*, 2015) adalah sebuah metode estimasi *value function* yang dapat bekerja tidak hanya untuk *state* tetapi juga untuk target atau *goal*. Konsep *value function* $V(s;\theta)$ pada subbab sebelumnya merujuk pada estimasi nilai ekspektasi *reward* maksimum yang dapat diperoleh pada *state* s hingga akhir episode berdasarkan *parameter* tertentu θ. Pada UVFA, konsep ini dikembangkan menjadi $V(s,g;\theta)$, yaitu melakukan estimasi nilai ekspektasi *reward* maksimum yang dapat diperoleh pada *state* s apabila hendak mencapai *goal* g yang diinginkan menggunakan sekumpulan *parameter* tertentu θ. Secara praktis, *value function* biasa hanya dapat digunakan untuk melakukan estimasi nilai *reward* untuk mencapai sebuah tujuan akhir, sedangkan UVFA dilatih untuk melakukan estimasi *reward* untuk setiap tujuan yang mungkin. UVFA memanfaatkan kemiripan struktur dari *state* s dan *goal* g untuk melakukan generalisasi sehingga selain mengurangi jumlah *state* yang harus dipelajari. UVFA juga telah dibuktikan dapat melakukan generalisasi terhadap target yang tidak pernah ditemui sebelumnya.

UVFA dapat diimplementasikan dengan membuat 2 buah *nonlinear function approximator* (ϕ,ψ) yang masing-masing berguna untuk melakukan *embedding state* dan *goal* (s,g). Hasil kedua *embedding* ini kemudian akan digabungkan untuk menjadi fitur yang menjadi dasar estimasi nilai *value function* $V(s,g;\theta)$. *Function approximator* ini kemudian dilatih dengan metode yang serupa dengan *Q learning* pada DQN, sehingga terbentuklah sebuah *regressor* yang dapat mengestimasi *value function* sebuah pasangan *state* dan *goal*. Rumus *update* nilai *value function* pada UVFA dapat dilihat pada persamaan 2.11 (Schaul *et al.*, 2015):

$$Q(s_t, a_t, g) \leftarrow \alpha \left(r_g + \gamma_g \max_{a'} Q(s_{t+1}, a', g) \right) + (1 - \alpha) Q(s_t, a_t, g) \qquad (2.11)$$

2. 11. *Hindsight Experience Replay*

Hindsight Experience Replay (HER) (Andrychowicz *et al.*, 2017) adalah sebuah teknik baru dalam *reinforcement learning* yang digunakan untuk mengatasi sebuah masalah dalam *reinforcement learning*, yaitu perbedaan perbandingan jumlah kondisi sukses dengan *reward* positif dan jumlah kondisi gagal dengan *reward* negatif (*sparse reward*). Hal ini dilakukan dengan cara memodifikasi metode pengumpulan pengalaman pada *replay memory buffer*. Penggunaan HER membuat tidak perlunya dilakukan *reward shaping* untuk mengatasi masalah *sparse reward* dan membuat algoritma *reinforcement learning* dapat digunakan untuk menyelesaikan masalah yang bersifat *multi-goal* (tujuan akhir yang diminta dapat dikustomisasi).

Inisialisasi algoritma *reinforcement learning off-policy* A
Inisialisasi *replay memory* R
For $episode = 1, M$ do
    Terima *goal* g dari *input* dan *state* awal s_0 dari *environment*
    For $t = 1, T$ do
        Pilih aksi a_t berdasarkan *policy* dari A
            $a_t \leftarrow \pi_b(s_t || g)$
        Lakukan aksi a_t dan terima *state* baru s_{t+1}
    End For
    For $t = 1, T$ do
        $r_t := r(s_t, a_t, g)$
        simpan transisi $(s_t||g,\ a_t,\ r_t,\ s_{t+1}||g)$ di R
        ambil beberapa sampel *goal* yang dapat terjadi G
        for $g' \in G$ do
            $r' := r(s_t, a_t, g')$
            simpan transisi $(s_t||g',\ a_t,\ r',\ s_{t+1}||g')$ di R
        End For
    End For
    For $t = 1, N$ do
        Sampel *random* minibatch transisi B dari R
        Lakukan 1 langkah optimasi pada A menggunakan B
    End For
End For

Algoritma 2.2 *Hindsight Experience Replay* (Andrychowicz *et al.*, 2017)

Pada metode biasa, *replay memory* hanya menyimpan data transisi biasa (s_t, a_t, r_t, s_{t+1}) dengan *reward* r_t positif apabila aksi yang dilakukan baik dan *reward* r_t negatif apabila aksi yang dilakukan buruk. HER memanfaatkan konsep UVFA, yaitu dengan menambahkan *goal* g pada transisi yang disimpan menjadi $(s_t, a_t, r_t, s_{t+1}, g)$ dan mengubah *reward* menjadi positif apabila aksi yang diambil pada *state* membuat *goal* tercapai dan *reward* negatif apabila aksi tidak menyebabkan *goal* tercapai. Selain menyimpan transisi $(s_t, a_t, r_t, s_{t+1}, g)$, disimpan juga transisi yang serupa tetapi mengganti *goal* dengan *state* yang sebenarnya dicapai $(s_t, a_t, r_t, s_{t+1}, s_{t+1})$ dengan *reward* positif. Secara intuisi, hal ini dapat diibaratkan bahwa agen dapat belajar dari kesalahan (agen gagal mencapai tujuan asli g tetapi agen mendapat pelajaran bahwa ketika *state* s_t dihadapi dan *state* s_{t+1} ingin dicapai, maka lakukan aksi a_t). Secara praktis hal ini dapat menyeimbangkan jumlah pengalaman dengan *reward* positif dan negatif yang diterima dan sekaligus dapat mempercepat generalisasi dari agen (karena penggunaan konsep UVFA). Algoritma HER dapat dilihat pada algoritma 2.2

2. 12. *Goal Conditioned Supervised Learning*

Goal Conditioned Supervised Learning (GCSL) (Ghosh *et al.*, 2019) merupakan sebuah teknik baru dalam multi-*goal reinforcement learning* yang memanfaatkan kestabilan proses pelatihan *imitation learning* dan *supervised learning* untuk diaplikasikan ke dalam multi-*goal reinforcement learning*. Salah satu hal yang membuat pelatihan *deep reinforcement learning* butuh waktu lama dan tidak stabil adalah karena model *neural network* berusaha memprediksi *value function* dengan target yang berubah-ubah berdasar pengalaman yang diperoleh. Dari persamaan 2.5 dan persamaan 2.11 dapat kita lihat bahwa untuk melakukan *update* Q *function*, diperlukan nilai Q *function* untuk s_t dan s_{t+1}, di mana kedua nilai ini diperoleh dari *neural network* yang sedang dilatih. Setiap kali *update* dilakukan, nilai Q *function* target juga dapat berubah, sehingga *neural network* berusaha melakukan regresi pada target yang bergerak (moving target).

GCSL menggunakan perspektif bahwa *reinforcement learning* proses memperoleh *policy* yang baik dari data *training* yang baik dan memperoleh data

training yang baik dari *policy* yang baik. Pembuatan *policy* yang baik dari data *training* yang baik biasa ditemukan dalam *supervised learning* atau dalam metode *reinforcement learning* yang bersifat *imitation learning*. Pada kedua metode ini, AI belajar dari data yang dilabeli oleh manusia (atau dari demonstrasi manusia), tidak dari hasil *trial and error*. Pembuatan data *training* yang baik dilakukan dengan menggunakan prinsip yang sama dengan HER, yaitu mengubah data yang buruk (gagal) menjadi data yang baik (berhasil) dengan cara mengubah *goal* yang dituju.

Arsitektur GCSL terdiri dari sebuah *neural network classifier* yang menerima *input* berupa *state* saat ini, *goal* yang telah dicapai saat ini, dan *goal* apa yang hendak dicapai. *Output* dari *network* ini adalah prediksi aksi apa yang harus diambil agar *goal* yang diminta dapat tercapai. *Neural network* ini dilatih dengan menggunakan data yang diperoleh dari hasil interaksi *agent* dengan *environment*. Data ini sebelumnya harus melalui proses *relabeling* agar data yang awalnya buruk menjadi data yang dapat digunakan untuk proses *training*.

Proses *relabeling* data pada GCSL mirip dengan proses *relabeling* data pada HER. Misalkan pada sebuah episode, *agent* berinteraksi dengan *environment* dan menghasilkan transisi:

$$\{(s_0, g_0, g, a_0), (s_1, g_1, g, a_1), (s_2, g_2, g, a_2), (s_3, g_3, g, a_3)\}$$

di mana s_t adalah *state* pada waktu t, a_t adalah aksi yang diambil saat t, g_t adalah *goal* yang berhasil dicapai saat t, dan g adalah *goal* yang ingin dicapai. Pada episode ini *goal* tidak berhasil dicapai, tetapi data transisi tersebut berhasil mencapai *goal* g_1, g_2, dan g_3. Dari keempat transisi ini akan dilakukan *relabeling* sehingga menjadi 6 data *training*, yaitu:

$$\{(s_0, g_0, g_1, a_0), (s_0, g_0, g_2, a_0), (s_0, g_0, g_3, a_0),$$
$$(s_1, g_1, g_2, a_1), (s_1, g_1, g_3, a_1), (s_2, g_2, g_3, a_2)\}$$

Dalam proses *relabeling* ini dapat dilihat bahwa data yang awalnya buruk dapat diubah menjadi data yang lebih baik.

Cara kerja GCSL adalah pertama-tama *agent* yang telah diinisialisasi awal secara *random* akan berinteraksi dengan *environment* untuk memperoleh data transisi. Pada data transisi ini kemudian akan dilakukan proses *relabeling goal* seperti yang dijelaskan sebelumnya sehingga menjadi data *training*. *Neural network*

kemudian dilatih dengan metode pelatihan *supervised learning classifier* menggunakan data *training* yang telah dibuat. Langkah ini diulangi terus menerus sesuai permintaan. Metode *supervised learning* yang dilakukan secara iteratif ini telah dibuktikan dapat menghasilkan data *training* yang lebih baik di setiap iterasinya karena perbaikan *policy* setiap kali *update* dilakukan. Ringkasan cara kerja GCSL dapat dilihat pada algoritma 2.3.

```
Inisialisasi policy π(s,g,g′)
Inisialisasi replay memory R
For episode = 1,M do
  Terima goal g dari input
  Terima state s_0 dan g_0 dari environment
  Inisialisasi sebuah temporary buffer B
  For t=0,T do
    Pilih aksi a_t berdasarkan policy dari A
          a_t ← π(s_t, g_t, g)
    Lakukan aksi a_t pada environment
    Terima state baru s_t+1 dan g_t+1
    Simpan transisi (s_t, g_t, g, a_t) di B
  End For
  R = {(s_t, g_t, g_(t+h), a_t):t, h > 0, t + h <= T}
  Update parameter ` menggunakan data dari R
End For
```

Algoritma 2.3 *Goal Conditioned Supervised Learning* (Ghosh *et al.*, 2019)

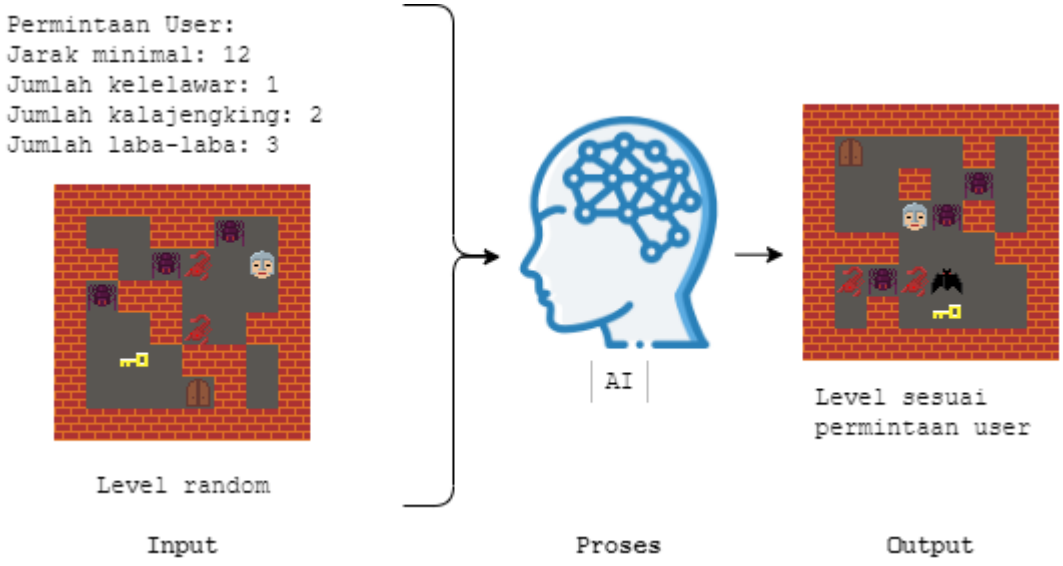
[*Halaman ini sengaja dikosongkan*]

BAB 3
METODOLOGI PENELITIAN

Bab ini akan memaparkan tentang metodologi penelitian yang akan dilakukan ini. Penelitian ini terdiri dari 5 tahap, yaitu studi literatur, perancangan dan implementasi metode, uji coba, analisa hasil, dan penyusunan laporan.

3. 1. Studi Literatur

Penelitian diawali dengan proses pengkajian yang berkaitan dengan topik penelitian yang diambil. Pada penelitian ini, referensi yang digunakan diperoleh dari jurnal yang memiliki hubungan dengan *procedural level generation*, *reinforcement learning*, *deep reinforcement learning*, *procedural level generation* menggunakan *reinforcement learning*, *hindsight experience replay*, dan *goal conditioned supervised learning*.



Gambar 3.1. Ilustrasi AI yang akan dibuat

3. 2. Perancangan dan Implementasi

Pada subbab ini akan dijelaskan mengenai perancangan penelitian yang hendak dilakukan. Tujuan dari penelitian ini adalah untuk menghasilkan sebuah agen yang dapat digunakan untuk membuat sebuah *level game* yang memenuhi semua kriteria yang diberikan oleh pengguna, bukan untuk membuat agen untuk

memainkan *game* Zelda. Ilustrasi agen yang hendak dibuat dapat dilihat pada Gambar 3.1.

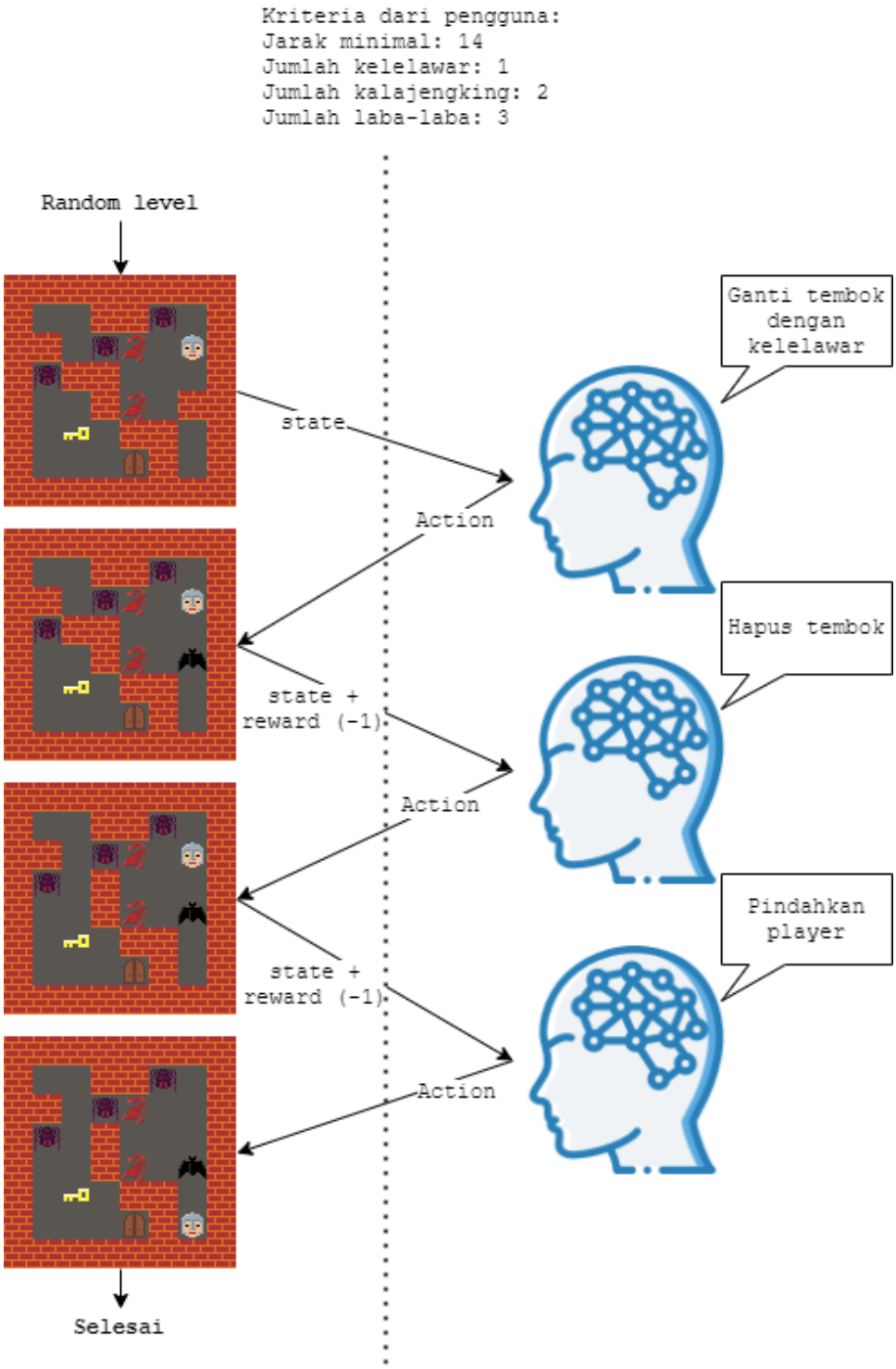
Pembuatan *level* dilakukan dengan cara melakukan modifikasi secara iteratif terhadap sebuah *level* hingga *level* tersebut dapat dimainkan dan memenuhi semua kriteria yang diminta oleh pengguna. Cara ini sama dengan cara yang digunakan pada (Khalifa *et al.*, 2020). Perbedaannya adalah pada penelitian sebelumnya, agen dianggap sukses ketika berhasil membuat sebuah *level* yang dapat dimainkan. Pada penelitian ini, ditambahkan unsur tambahan yaitu pengguna dapat meminta kriteria-kriteria tertentu yang harus dipenuhi oleh *level* yang dihasilkan.

Karena pada penelitian ini agen harus dapat memenuhi permintaan dari pengguna, algoritma *reinforcement learning* biasa yang digunakan pada (Khalifa *et al.*, 2020) tidak dapat digunakan di sini. Salah satu metode yang dapat digunakan adalah algoritma multi-*goal reinforcement learning* seperti pada (Andrychowicz *et al.*, 2017; Ghosh *et al.*, 2019). Algoritma multi-*goal reinforcement learning* pada penelitian sebelumnya biasanya diimplementasikan untuk melatih lengan robot untuk menggerakkan tangannya ke posisi yang diminta oleh pengguna.

Penelitian ini dapat dianggap sebagai salah satu upaya untuk menggabungkan *level generator* pada (Khalifa *et al.*, 2020) dengan kemampuan untuk memenuhi permintaan pengguna pada (Andrychowicz *et al.*, 2017; Ghosh *et al.*, 2019). Terminologi dan implementasi dari setiap komponen pada penelitian ini akan banyak mengambil inspirasi dari *source code* yang mengimplementasi hasil penelitian (Andrychowicz *et al.*, 2017; Ghosh *et al.*, 2019; Khalifa *et al.*, 2020) dan dari contoh *library reinforcement learning stable-baselines* (Hill *et al.*, 2018).

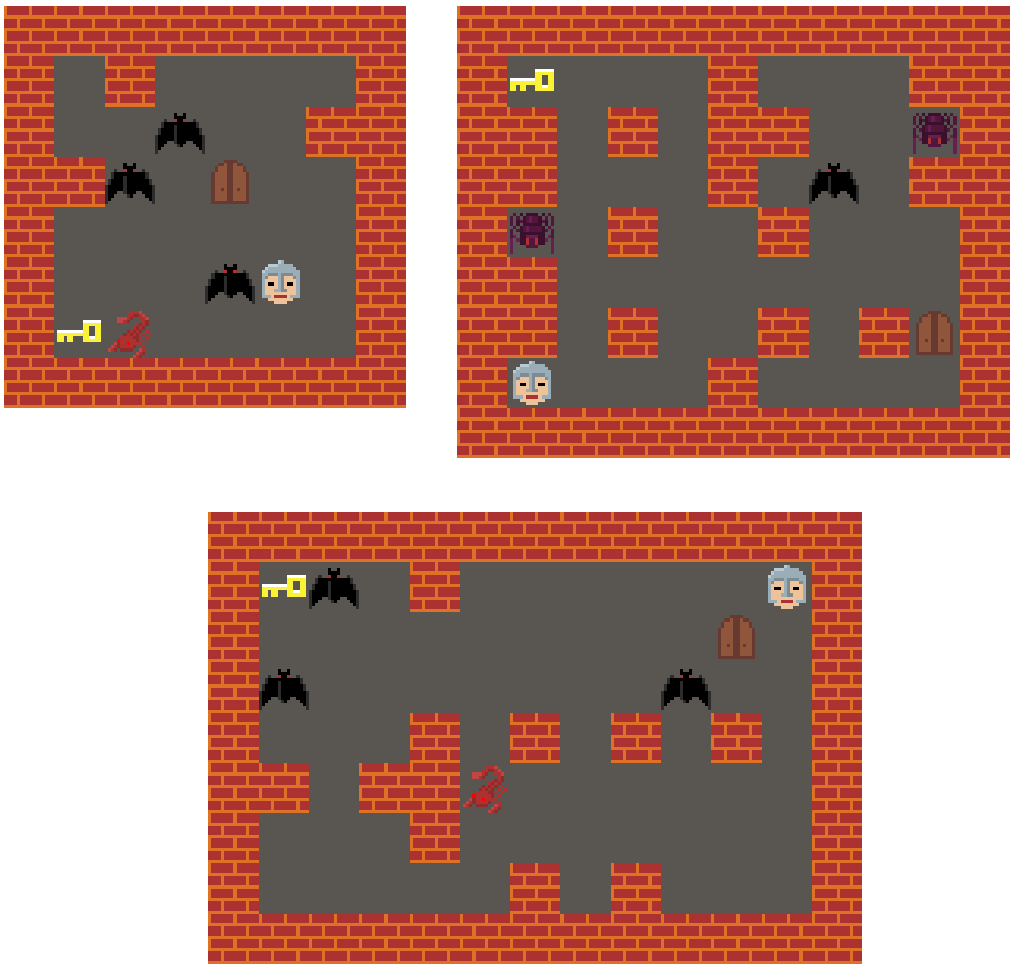
Untuk membuat sebuah *level*, pertama-tama program akan meminta kriteria-kriteria apa yang harus dipenuhi oleh *level* yang dihasilkan. Kemudian, program akan membuat sebuah *level* secara *random*. Agen kemudian akan menerima *input* berupa *level random* tadi dan kriteria dari pengguna. Setelah itu, agen akan memberikan sebuah modifikasi terbaik yang dapat dilakukan untuk *level* tersebut. Apabila *level* yang telah dimodifikasi masih belum memenuhi kriteria yang diminta, *level* hasil ini akan dijadikan *input* untuk iterasi berikutnya. Kedua langkah terakhir ini akan diulangi terus menerus hingga *level* yang dihasilkan memenuhi semua permintaan pengguna. Pada penelitian ini, kondisi ketika *level* yang

dihasilkan memenuhi semua permintaan pengguna ditandai dengan *reward* 0 sedangkan kondisi belum memenuhi ditandai dengan *reward* -1. Contoh ilustrasi proses pembuatan *level* ini dapat dilihat pada Gambar 3.2.



Gambar 3.2. Ilustrasi proses pembuatan *level*

Dilihat dari Gambar 3.2, dibutuhkan 2 komponen penting untuk dapat membuat agen ini. Komponen pertama yang diperlukan adalah sebuah *environment*. *Environment* bertugas untuk mengevaluasi sebuah *level* (apakah sudah memenuhi kriteria pengguna atau belum) dan memproses perintah dari agen untuk memodifikasi *level*. Pada Gambar 3.2, *environment* menangani semua proses yang terjadi pada bagian kiri garis.



Gambar 3.3. Contoh *level* pada *game* Zelda untuk penelitian ini

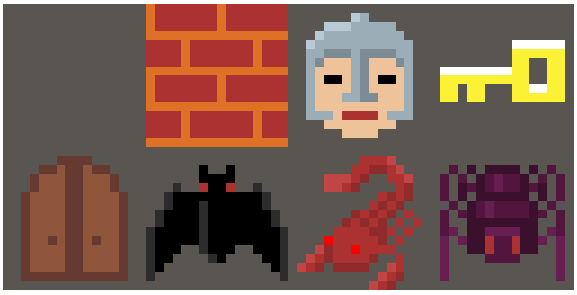
Komponen kedua yang diperlukan adalah agen yang berupa sebuah arsitektur *neural network* dengan *hyperparameter* yang cocok. *Neural network* ini dilatih untuk dapat menentukan modifikasi terbaik untuk *level* yang diberikan. Modifikasi yang terbaik di sini maksudnya adalah modifikasi yang membuat *level*

semakin mendekati permintaan dari *user*. Pada Gambar 3.2, agen *neural network* menangani semua proses yang terjadi pada bagian kanan garis.

Pada subbab ini akan dijelaskan tentang detail dari implementasi komponen-komponen di atas. Subbab ini akan dibagi menjadi 4 bagian. Bagian pertama akan membahas tentang detail *game* Zelda yang akan digunakan. Bagian kedua akan membahas cara kerja *environment* yang akan digunakan. Bagian ketiga akan membahas tentang detail *input*, *output*, dan arsitektur *neural network* yang akan digunakan, serta cara arsitektur *neural network* memproses *input* dan menghasilkan *output*. Bagian keempat akan membahas proses pelatihan *neural network* sehingga dapat menentukan modifikasi terbaik.

3.2.1 *Game* Zelda

Penelitian ini menggunakan *game* Zelda versi GVGAI (Perez-Liebana *et al.*, 2019) sebagai *game* utama yang hendak dibuat *level*nya. Pada *game* Zelda, pemain menang dengan cara menggerakkan karakter utama untuk mengambil kunci dan keluar dari *level* melewati pintu. Terdapat beberapa musuh yang bergerak secara *random* di *level*. Apabila karakter bertabrakan dengan musuh maka pemain akan kalah. Contoh 3 buah *level* dari *game* Zelda ini dapat dilihat pada Gambar 3.3.



Gambar 3.4. Delapan jenis *tile* dari kiri atas ke kanan dan bawah: *empty*, *solid*, *player*, *key*, *door*, *bat*, *scorpion*, dan *spider*

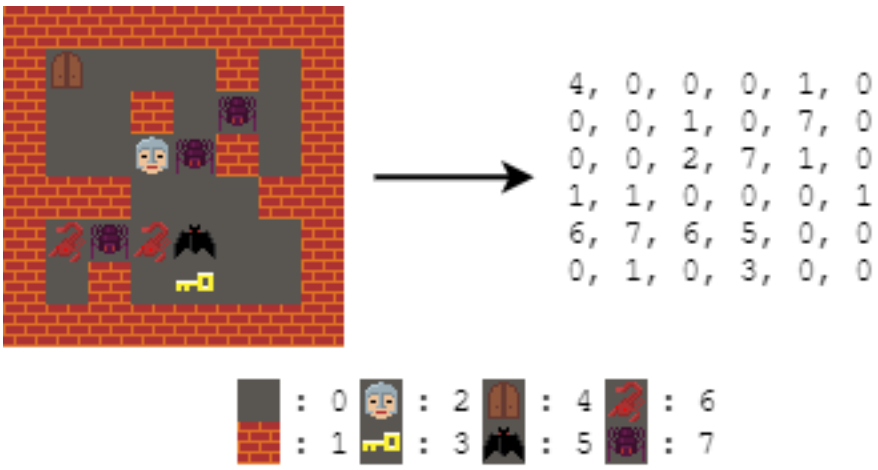
Sebuah *level* pada *game* Zelda terdiri dari sebuah *grid* yang berisi beberapa jenis *tile* (petak). Terdapat 8 jenis petak dalam *game* ini. Gambar dari masing-

masing petak dapat dilihat pada Gambar 3.4. Berikut adalah penjelasan masing-masing petak:

- *Empty* (kosong): petak kosong yang dapat dilewati oleh karakter atau musuh. Petak ini direpresentasikan dengan angka 0.
- *Solid* (tembok): petak yang berisi tembok tidak dapat dilewati oleh karakter atau musuh. Petak ini direpresentasikan dengan angka 1.
- *Player* (karakter): karakter yang dikendalikan oleh pemain. Pemain dapat menggerakkan karakter ke atas, bawah, kiri, dan kanan 1 petak. Karakter juga dapat menggunakan pedang untuk membunuh musuh yang berada di depannya. Musuh yang dibunuh akan hilang dari *level* sehingga petak yang ditempati musuh sekarang menjadi bisa dilewati. Karakter harus digerakkan untuk mengambil kunci kemudian masuk ke pintu untuk memenangkan *level*. Jika karakter bertabrakan dengan musuh maka *level* akan berakhir dan pemain kalah. Petak ini direpresentasikan dengan angka 2.
- *Key* (kunci): objek ini harus diambil terlebih dahulu agar *player* boleh memasuki pintu untuk memenangkan *level*. Setelah kunci diambil maka petak ini akan menjadi petak kosong. Petak ini direpresentasikan dengan angka 3.
- *Door* (pintu): petak yang merupakan tujuan akhir dari pemain. Pemain akan menang apabila telah menggerakkan karakternya untuk mengambil kunci dan masuk ke pintu. Petak ini direpresentasikan dengan angka 4.
- *Bat* (kelelawar): salah satu jenis musuh pada *game*. Apabila musuh menyentuh karakter maka pemain akan kalah. Musuh dapat dihilangkan dengan cara karakter berdiri di sebelah musuh dan menghadap musuh kemudian menggunakan pedang. Setelah musuh dihilangkan maka petak itu menjadi petak kosong yang dapat dilewati. Kelelawar bergerak ke arah *random* setiap 2 detik sekali. Petak ini direpresentasikan dengan angka 5.
- *Scorpion* (kalajengking): salah satu jenis musuh pada *game*. Kalajengking bergerak ke arah *random* setiap 8 detik sekali. Petak ini direpresentasikan dengan angka 6.
- *Spider* (laba-laba): salah satu jenis musuh pada *game*. laba-laba bergerak ke arah *random* setiap 4 detik sekali. Petak ini direpresentasikan dengan angka 7.

Agar sebuah *level* pada *game* Zelda dapat dimainkan, terdapat beberapa syarat (*constraint*) yang harus terpenuhi. Syarat pertama adalah jumlah *player*, kunci, dan pintu pada sebuah *level* harus 1. Syarat berikutnya adalah kunci dan pintu harus dapat dicapai oleh karakter. Kunci dan pintu bisa saja tidak dapat dicapai oleh karakter jika jalan menuju kunci atau pintu terhalang oleh tembok. Musuh tidak dianggap dapat menghalangi jalan karakter karena musuh secara periodik dapat berpindah tempat atau dapat dihilangkan dari *level* dengan cara dibunuh menggunakan pedang karakter.

Sebuah *level game* Zelda pada penelitian ini direpresentasikan sebagai sebuah *array* 2 dimensi. Dimensi pertama dan kedua dari *array* merepresentasikan posisi koordinat y dan x dari *grid level*. Isi dari masing-masing elemen *array* adalah sebuah angka yang merepresentasikan *tile* apa yang terletak pada posisi tersebut. *Tile* yang direpresentasikan oleh masing-masing angka telah dijelaskan di paragraf sebelumnya. Implementasi ini serupa dengan (Khalifa *et al.*, 2020). Contoh representasi sebuah *level game* Zelda dapat dilihat pada Gambar 3.5. Pada penelitian ini digunakan 3 ukuran *level* sebagai perbandingan, yaitu 6×6, 7×9, dan 7×11.



Gambar 3.5. Contoh representasi sebuah *level* dalam bentuk *array* 2 dimensi

Dari aturan *game* yang telah dijelaskan, kita dapat menyimpulkan bahwa mudah atau sulitnya sebuah *level* diselesaikan ditentukan oleh 3 faktor utama: jenis

musuh yang ada pada *level*, banyaknya musuh yang ada pada *level*, dan jarak minimum yang harus ditempuh karakter untuk dapat mencapai kunci dan pintu. Musuh yang bergerak cepat lebih sulit dihindari sehingga memperbesar peluang pemain untuk kalah. Jumlah musuh yang banyak membuat pemain lebih sering bertemu musuh sehingga memperbesar peluang pemain untuk kalah. Semakin jauh jarak minimal yang harus ditempuh maka semakin besar pula peluang pemain bertemu dengan musuh dalam perjalanan menuju kunci dan pintu.

Dari kesimpulan ini ditentukan bahwa dalam penelitian ini akan ada 4 komponen *game* yang dapat diatur memanipulasi tingkat kesulitan *level* yang hendak dibuat. Komponen-komponen tersebut ialah banyaknya jumlah kelelawar, kalajengking, dan laba-laba pada *level*, serta jarak minimum yang harus ditempuh karakter untuk menang. Dalam penelitian ini, komponen-komponen yang dapat dimanipulasi ini akan disebut sebagai *goal* (Andrychowicz *et al.*, 2017; Hill *et al.*, 2018; Ghosh *et al.*, 2019).

Sebuah *goal* untuk *game* Zelda pada penelitian ini direpresentasikan sebagai *tuple* 5 angka, misalnya (12, 1, 2, 3, 1). Berikut adalah arti dari masing-masing angka dalam *goal*:

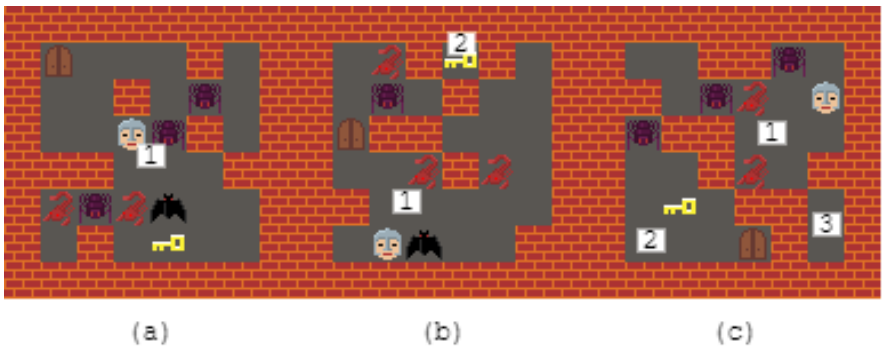
¶ Angka pertama merepresentasikan jarak minimum yang harus ditempuh karakter untuk menang. Jarak minimum dihitung dari berapa banyaknya langkah minimum yang diperlukan oleh karakter untuk dapat berjalan mencapai kunci dilanjutkan dengan mencapai pintu. Jarak ini dihitung dengan menggunakan algoritma Dijkstra. Apabila pintu atau kunci terhalang oleh tembok (level tidak bisa diselesaikan) maka jarak minimum akan diisi dengan angka 0. Contoh *level* dengan kunci atau pintu yang tidak dapat dicapai oleh karakter dapat dilihat pada Gambar 3.6b dan Gambar 3.6c. Jarak minimum yang harus ditempuh untuk menang yang boleh dipilih pada penelitian ini dibatasi hanya pada jangkauan tertentu. Jangkauan ini berbeda untuk masing-masing ukuran *level*. Untuk *level* berukuran 6×6, ditentukan jarak minimal yang boleh dipilih adalah 2 - 16 langkah. Untuk *level* 7×9 minimalnya 2 - 29 langkah, sedangkan untuk *level* 7×11 minimalnya 2 - 36 langkah. Karena jarak minimal paling kecil yang dapat dipilih adalah 2 maka pada *level* yang

dihasilkan, karakter pasti dapat mencapai kunci dan pintu sehingga *level* pasti dapat dimainkan.

¶ Angka kedua merepresentasikan jumlah kelelawar pada *level*. Pada penelitian ini, jumlah kelelawar pada sebuah *level* dibatasi hanya boleh ada 0 sampai 3.

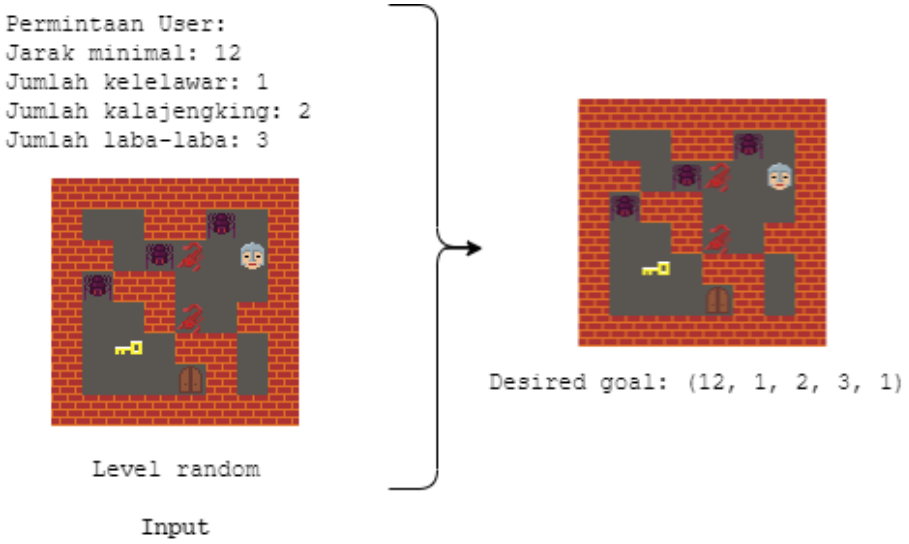
¶ Angka ketiga merepresentasikan jumlah kalajengking pada *level*. Pada penelitian ini, jumlah kalajengking pada sebuah *level* dibatasi hanya boleh ada 0 sampai 3.

¶ Angka keempat merepresentasikan jumlah laba-laba pada *level*. Pada penelitian ini, jumlah laba-laba pada sebuah *level* dibatasi hanya boleh ada 0 sampai 3.



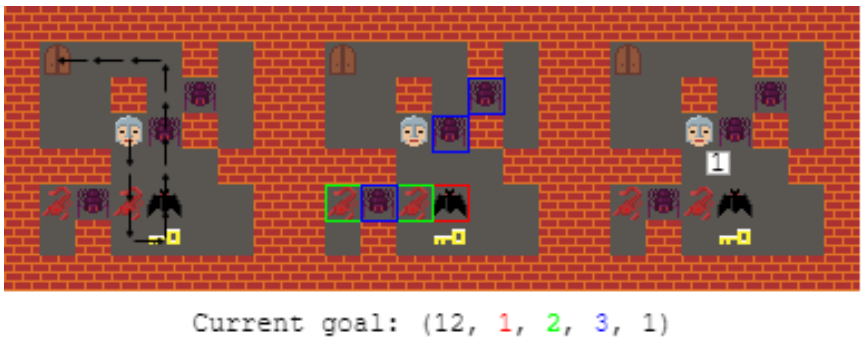
Gambar 3.6. Contoh 3 buah *level* dan jumlah *region* pada masing-masing *level*.

¶ Angka kelima merepresentasikan banyaknya *region* dalam sebuah *level*. Sebuah *region* pada sebuah *level* didefinisikan sebagai sebuah himpunan petak pada *level* yang dapat dikunjungi (semua petak selain tembok) oleh *player* apabila *player* berada pada salah satu dari petak dalam himpunan tersebut. Apabila divisualisasikan, sebuah *level* yang memiliki 1 *region* terlihat memiliki sebuah ruangan. *Level* yang memiliki lebih dari 1 *region* terlihat seperti terdiri dari beberapa ruangan yang saling terisolasi sehingga *player* tidak dapat mengunjungi ruangan lain tersebut (buntu). Contoh visualisasi *region* dapat dilihat pada Gambar 3.6. Gambar 3.6a merupakan contoh *level* yang memiliki 1 *region*, Gambar 3.6b 2 *region*, dan Gambar 3.c memiliki 3 *region*. Hanya petak tembok yang dapat menghalangi jalannya *player* (musuh dapat dihilangkan dengan cara dibunuh).



Gambar 3.7. Ilustrasi desired goal

Pada penelitian ini akan ada 2 buah istilah *goal* yang akan digunakan. Pertama, kriteria-kriteria yang diminta oleh pengguna yang harus dipenuhi oleh *level* setelah proses iterasi selesai disebut dengan *desired goal*. Ilustrasi dari *desired goal* dapat dilihat pada Gambar 3.7. Yang kedua, kriteria-kriteria yang dipenuhi oleh *level* saat ini disebut dengan *current goal*. Ilustrasi dari *current goal* dapat dilihat pada Gambar 3.8.

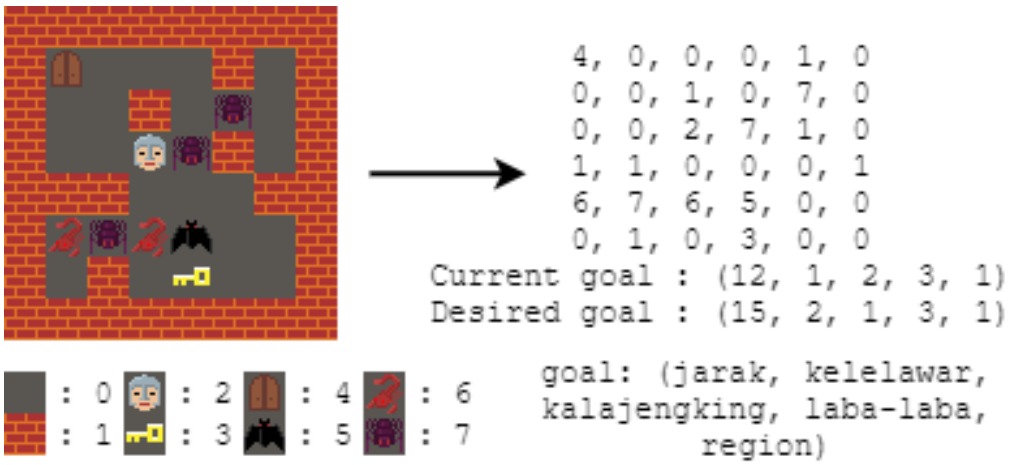


Gambar 3.8. Ilustrasi dari *current goal*

Sebuah *state* untuk *environment* Zelda terdiri dari sebuah representasi *level* dan *current goal*. Representasi *level* ini berupa *array* 2 dimensi yang menggambarkan kondisi *level* saat ini. *Current goal* berupa *tuple* 5 angka yang merepresentasikan

jarak minimum, jumlah kelelawar, kalajengking, laba-laba, dan *region* pada *level* saat ini. Contoh dari sebuah *state* yang akan digunakan dalam penelitian ini dapat dilihat pada Gambar 3.9.

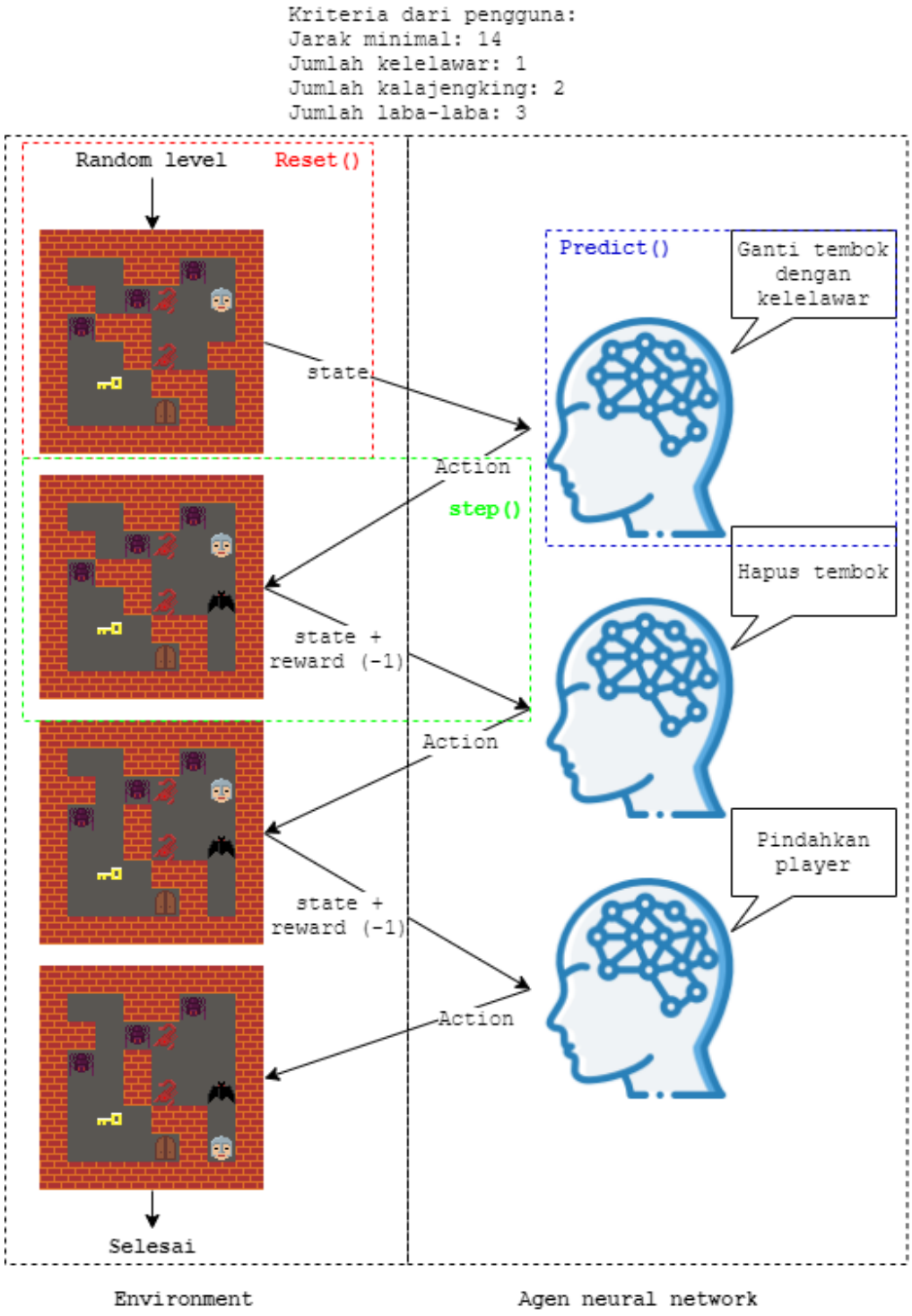
Pada gambar 3.9 dapat dilihat sebuah *level* Zelda dan contoh apabila *level* tersebut diubah menjadi sebuah representasi *state*. Representasi *state* berisi nilai-nilai sesuai dengan isi *tile* pada *grid level*. *Current goal* berisi $(12, 1, 2, 3, 1)$, sesuai dengan langkah minimum, banyaknya kelelawar, kalajengking, dan laba-laba pada *level*.



Gambar 3.9. Contoh sebuah state untuk game Zelda berukuran 6×6

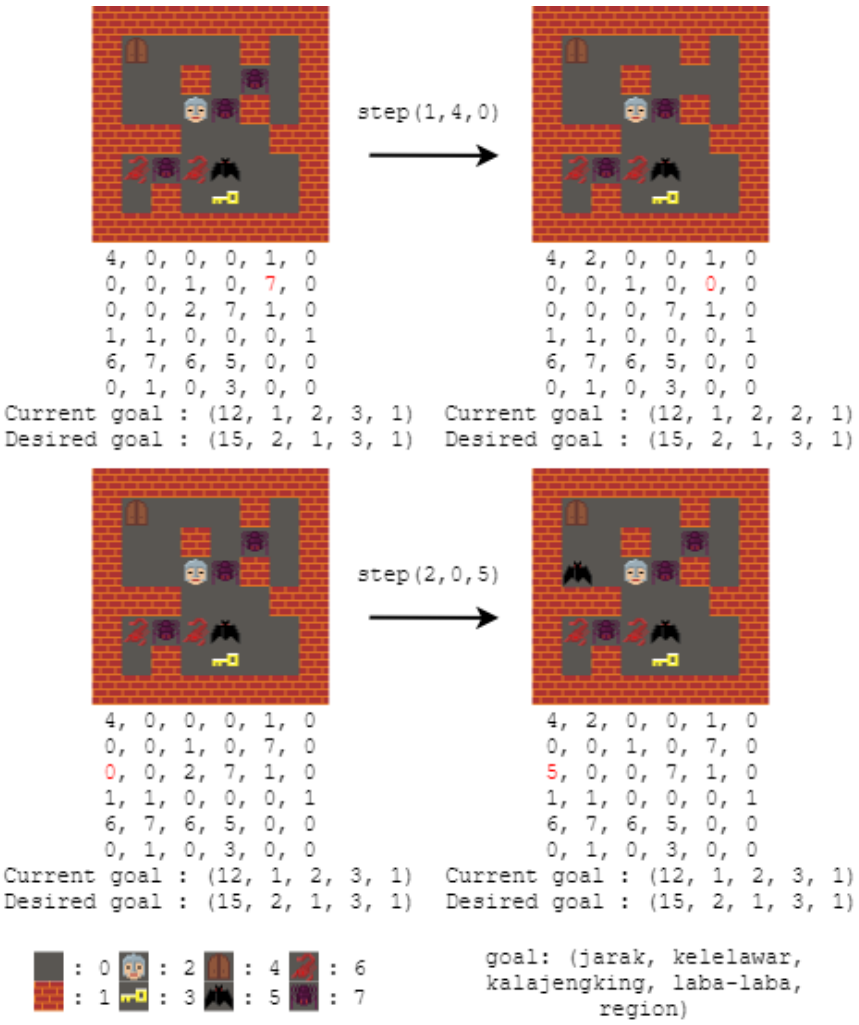
3.2.2 *Environment*

Environment pada *reinforcement learning* adalah komponen yang bertugas untuk memberikan *state* untuk agen, menerima aksi dari agen, dan memberi umpan balik berupa *reward* untuk agen. *Environment* pada penelitian ini dibuat mirip dengan penelitian (Khalifa *et al.*, 2020). Pada (Khalifa *et al.*, 2020), *environment* bertugas untuk membuat *level random* sebagai inisialisasi awal proses pembuatan *level*, memodifikasi *level* sesuai dengan perintah dari agen, dan memberi *reward* berdasarkan evaluasi *level* setelah dimodifikasi oleh agen. *Environment* akan memberi *reward* positif apabila *level* menjadi lebih baik dari sebelumnya dan *reward* negatif apabila *level* menjadi lebih buruk daripada sebelumnya.



Gambar 3.10. Ilustrasi interaksi antara *environment* dengan agen dalam membuat sebuah *level*

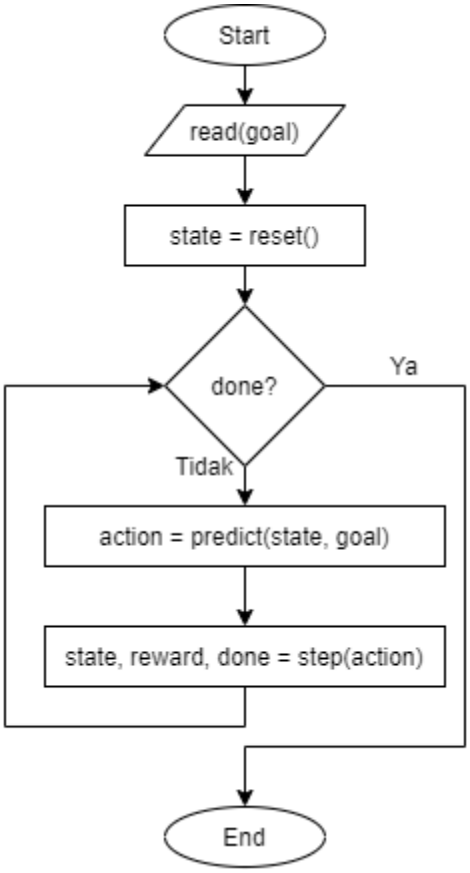
Pada penelitian ini diberikan 2 tambahan untuk *environment reinforcement learning* yang digunakan. Tambahan pertama adalah *environment* dapat menentukan *current goal* dari *level* saat ini. Tambahan kedua adalah *environment* dapat menentukan *reward* berdasarkan *state* dan *goal* yang diberikan. Kedua modifikasi ini diperlukan agar *environment* dapat digunakan untuk multi-*goal reinforcement learning*. Implementasi ini terinspirasi dari (Andrychowicz *et al.*, 2017; Hill *et al.*, 2018).



Gambar 3.11. Contoh eksekusi fungsi *step*

Ilustrasi interaksi antara *environment* dengan agen *level generator* dalam membuat sebuah *level* baru dapat digambarkan seperti pada Gambar 3.10. Pertama-tama *environment* membuat sebuah *level* baru secara *random* dengan mengeksekusi

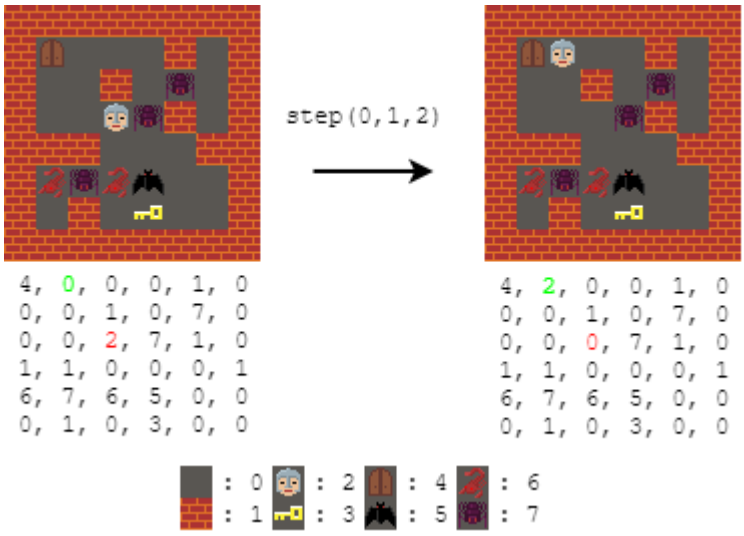
perintah `reset()`. Setelah itu akan dilakukan sebuah perulangan. Agen akan menentukan *action*, yaitu modifikasi terbaik yang dapat dilakukan pada *level* berdasarkan *state* yang diterima, dengan mengeksekusi perintah `predict(state, goal)`. *Action* ini kemudian akan diproses oleh *environment* dengan mengeksekusi fungsi `step(action)`. Fungsi ini kemudian akan menghasilkan *state* baru yang berisi *level* hasil modifikasi, *reward* untuk menentukan apakah *goal* telah terpenuhi, dan menentukan apakah perulangan perlu dihentikan.



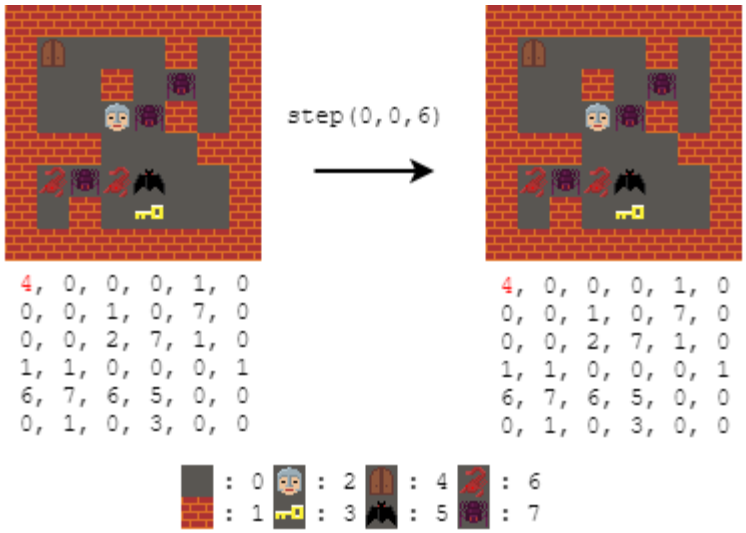
Gambar 3.12. Diagram alir penggunaan *environment*

Fungsi *step* dieksekusi ketika agen hendak melakukan modifikasi terhadap *level* saat ini. Fungsi *step* diimplementasikan sesuai dengan (Khalifa *et al.*, 2020). Fungsi ini memerlukan *parameter* berupa posisi x dan y yang hendak diubah dan nilai *tile* baru untuk posisi tersebut dalam bentuk (x, y, t). *Environment* kemudian

akan mengubah isi *tile* koordinat (y, x) dengan nilai *tile* baru sesuai *input*, mengkalkulasi ulang *current goal* dan menghitung *reward*. Setelah proses *step* dijalankan, *environment* akan mengembalikan *state* baru, *reward* yang diperoleh, dan apakah satu episode telah selesai. Gambar 3.11 menunjukkan contoh dari eksekusi fungsi *step*. Pada gambar pertama dapat dilihat bahwa *tile* baris 1 kolom 4 diganti dengan kosong (0) dan pada gambar kedua *tile* baris 2 kolom 0 diganti dengan kelelawar (5).



(a)



(b)

Gambar 3.13. Contoh implementasi dari batasan pertama untuk *environment*.

Untuk penelitian ini, sebuah episode dianggap selesai apabila *desired goal* berhasil dicapai (*reward* = 0) atau apabila fungsi *step* telah dipanggil melebihi batas yang telah ditentukan. Batas pada penelitian ini diatur sesuai dengan banyaknya *tile* pada *level* (contohnya untuk *level* berukuran 6×6, jumlah pemanggilan dibatasi hanya sampai 36 kali sejak *reset*). Gambar 3.12 berisi diagram alir yang menggambarkan penggunaan *environment* dalam penelitian ini.

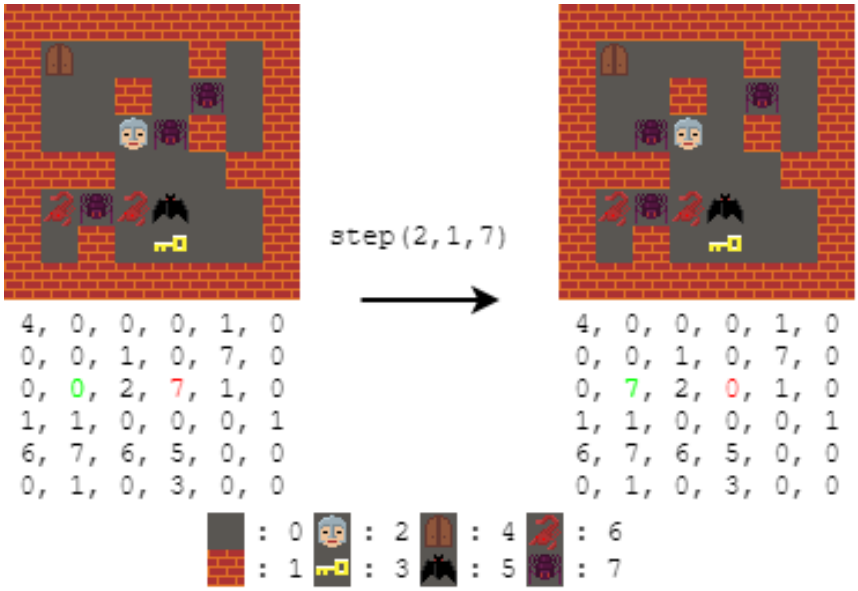
Pada penelitian ini diterapkan beberapa *constraint* (batasan) untuk mengurangi besarnya *state space* agar proses pelatihan dapat berjalan dengan lebih cepat atau untuk membuat agar *level* menjadi lebih baik. Berikut adalah daftar batasan yang diimplementasikan:

¶ Jumlah karakter, kunci, dan pintu pada *level* dibatasi harus berjumlah 1. Batasan ini dibuat agar agen tidak perlu mempelajari dari awal bahwa dalam sebuah *level* harus terdapat tepat 1 karakter, kunci, dan pintu, tidak boleh lebih, tidak boleh kurang. Implementasi dari batasan ini adalah pertama-tama *tile* karakter, kunci, dan pintu pada *grid* tidak dapat diganti dengan *tile* lain. Implementasi kedua adalah perintah *step* untuk *tile* karakter, kunci, dan pintu tidak hanya mengganti isi *tile* pada koordinat (y, x) dengan *tile* yang diminta tetapi juga menghapus *tile* itu di posisi yang lama (sehingga perintah ini berubah dari ubah isi *tile* menjadi pindah objek ke posisi baru). Contoh implementasi batasan ini dapat dilihat pada Gambar 3.13a dan 3.13b. Pada Gambar 3.13a dapat dilihat bahwa *tile* baris 0 kolom 1 diganti dengan *player* (2) dan *tile* baris 2 kolom 2 yang sebelumnya terisi *player* sekarang berisi kosong (0). Pada Gambar 3.13b dapat dilihat bahwa perintah $(0,0,6)$ diabaikan karena pada posisi baris 0 kolom 0 terdapat *tile* pintu.

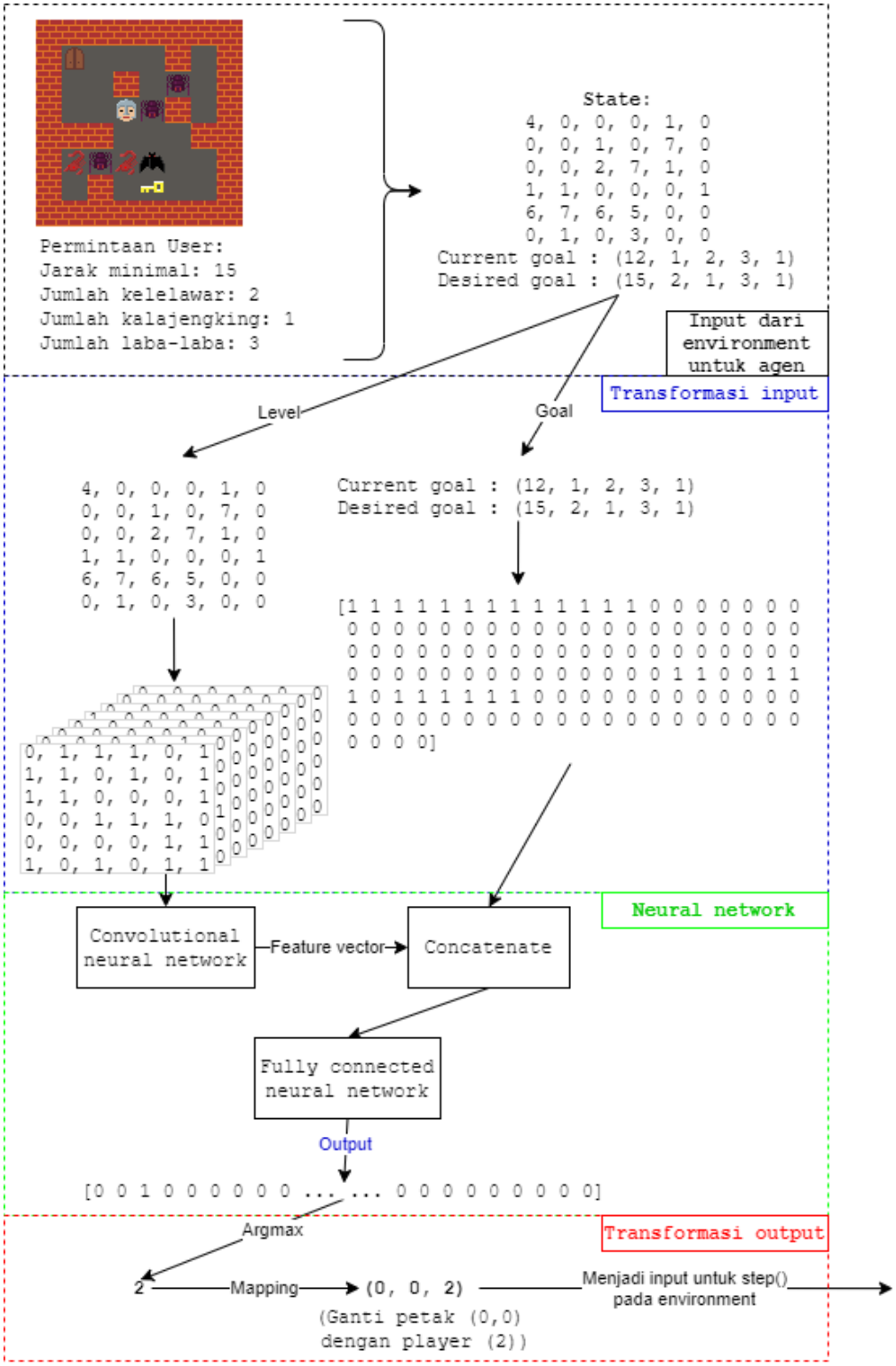
¶ Jumlah masing-masing jenis musuh dibatasi tidak boleh lebih dari 3. Misalnya, dalam sebuah *level* boleh ada 2 kelelawar dan 2 kalajengking (4 musuh) tapi tidak boleh ada 4 laba-laba (lebih dari 3). Batasan ini dibuat agar agen tidak perlu mempelajari dari awal bahwa jumlah maksimum masing-masing jenis musuh adalah 3. Implementasi dari batasan ini adalah jika jumlah sebuah jenis musuh pada *level* lebih dari 3, maka *environment* akan menghapus 1 musuh dengan jenis yang sama secara acak dari *level*. Contoh implementasi batasan

ini dapat dilihat pada Gambar 3.14. Pada Gambar 3.14 dapat dilihat bahwa *tile* baris 2 kolom 1 diganti dengan laba-laba (7) dan *tile* baris 2 kolom 3 yang sebelumnya terisi laba-laba sekarang berisi kosong (0).

¶ Setiap *tile* yang bukan tembok harus saling terhubung satu sama lain secara langsung atau tidak langsung. Pada kode program, kondisi ini disebut sebagai "memiliki 1 *region*". Tujuan dari batasan ini adalah untuk memaksa agen tidak membuat sebuah *level* dengan sebuah ruangan yang tidak dapat diakses (lebih dari 1 *region*). Apabila masih terdeteksi ada ruangan yang tidak dapat diakses maka *environment* akan menganggap jarak minimum *level* tersebut adalah 0. Hal ini secara tidak langsung menunjukkan pada agen bahwa *level* yang dibuat tidak bisa digunakan. Jumlah *region* pada sebuah *level* dicatat di *current goal* dan *desired goal* untuk proses *training*. Jumlah *region* pada *desired goal* diatur untuk selalu berjumlah 1.



Gambar 3.14. Contoh implementasi dari batasan kedua untuk *environment*.



Gambar 3.15. Ilustrasi proses *predict*

3.2.3 Arsitektur *Neural Network*

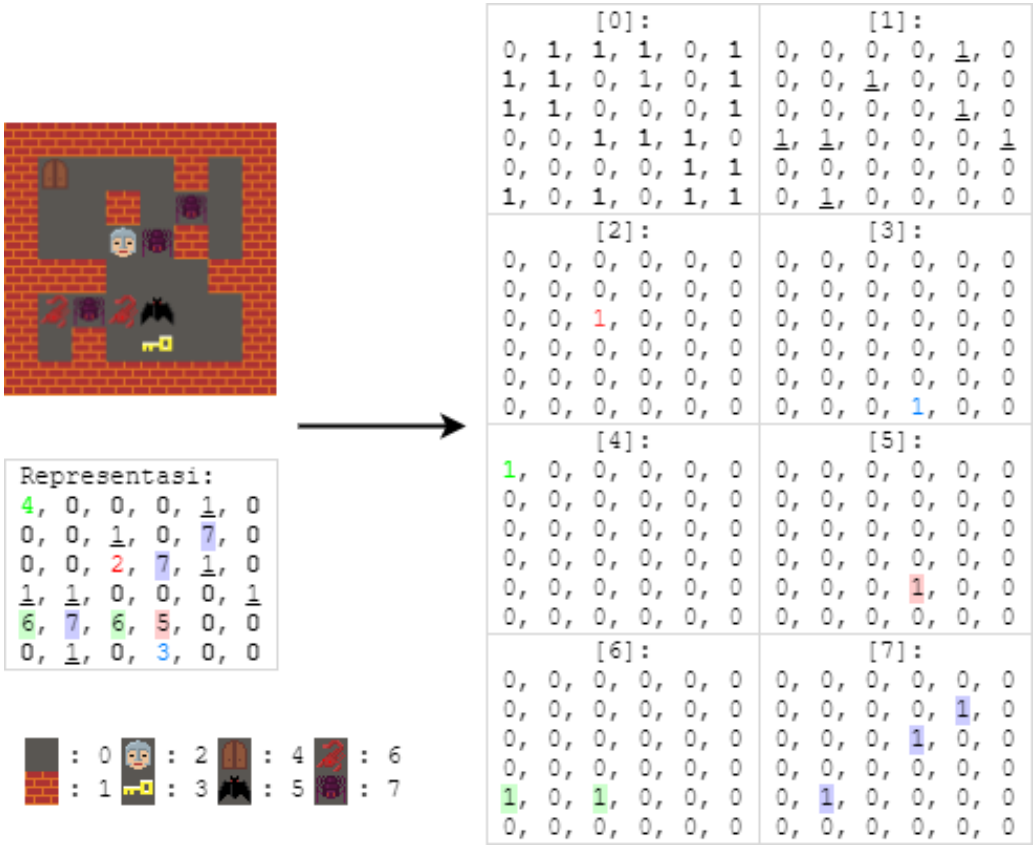
Pada subbab ini akan dijelaskan tentang arsitektur *neural network* yang akan digunakan. Akan dijelaskan pemisahan labeling yang dilakukan untuk *input*, arsitektur *neural network*, dan interpretasi dari *output* yang dihasilkan oleh *neural network*. Penelitian ini menggunakan 2 arsitektur yang berbeda, yaitu arsitektur DQN (Mnih *et al.*, 2013) dilengkapi dengan HER (Andrychowicz *et al.*, 2017) dan arsitektur GCSL (Ghosh *et al.*, 2019). Pada subbab ini beberapa contoh yang diberikan akan menggunakan *level* yang berukuran 6×6 agar contoh lebih mudah digambarkan. Detail untuk implementasi masing-masing ukuran (6×6, 7×9, dan 7×11) akan dipaparkan pada bagian uji coba.

Agen pada *reinforcement learning* adalah komponen yang belajar untuk dapat memperoleh *reward* maksimum dari hasil interaksinya dengan *environment*. Pada penelitian ini, agen bertugas untuk mempelajari modifikasi terbaik apa yang dapat dilakukan untuk memperbaiki *level* yang diberikan (Khalifa *et al.*, 2020). Agen ini dibuat dalam rupa sebuah arsitektur *neural network* yang menerima *input* berupa representasi *level*, *current goal*, dan *desired goal*, kemudian menghasilkan aksi yang terbaik untuk dapat mencapai *goal* (Andrychowicz *et al.*, 2017; Ghosh *et al.*, 2019). Agen akan dilatih menggunakan metode propagasi balik (backpropagation) menggunakan data *training* yang diperoleh dari hasil interaksi agen dengan *environment*.

Peran agen pada diagram di Gambar 3.10 terletak pada fungsi *predict*. Proses *predict* ini dapat dibagi lagi menjadi beberapa bagian yang dapat dilihat pada Gambar 3.15. Pertama-tama perlu dilakukan pemisahan labeling pada representasi *state* karena representasi *state* dari *environment* kurang cocok untuk menjadi *input neural network*. Hasil dari pemisahan labeling ini kemudian akan menjadi *input* untuk *neural network*. *Neural network* kemudian akan menghasilkan *output* berupa *array* 1 dimensi yang akan diinterpretasikan menjadi aksi terbaik untuk *state* tersebut. Aksi ini kemudian akan dieksekusi pada fungsi *step environment* untuk menghasilkan *level* yang lebih baik.

Transformasi representasi *state* dibagi menjadi 2, yaitu pemisahan labeling representasi *level* dan pemisahan labeling representasi *goal*. Metode pemisahan labeling representasi *level* yang digunakan sama dengan (Khalifa *et al.*, 2020).

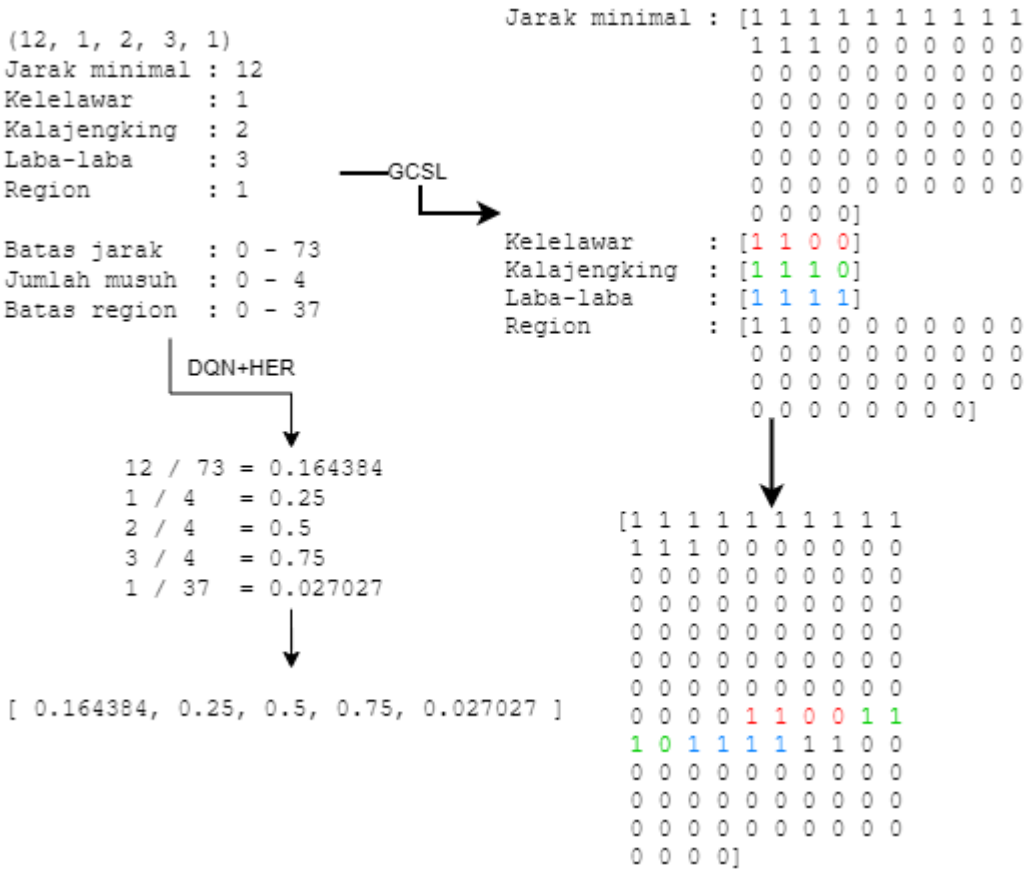
Representasi map yang berupa *array* 2 dimensi berisi bilangan bulat 0-7 dikonversi terlebih dahulu menjadi *array* 3 dimensi yang berisi 0 atau 1. Pada representasi *level* yang baru, dimensi pertama dari *array* menyatakan jenis dari *tile*. Dimensi kedua dan ketiga menyatakan posisi koordinat y dan x dari masing-masing *tile* tersebut. Hal ini berarti pada representasi baru, *array* indeks 0 akan berisi sebuah *array* 2 dimensi yang berisi 1 apabila pada posisi tersebut terdapat *tile* kosong / *empty* (0) dan 0 untuk posisi yang tidak berisi *tile* kosong. *Array* indeks 1 akan berisi sebuah *array* 2 dimensi yang berisi 1 apabila pada posisi tersebut terdapat *tile* tembok / *solid* (1) dan 0 untuk posisi yang tidak berisi *tile* tembok, dan seterusnya. Contoh pemisahan labeling untuk representasi *level* dapat dilihat pada Gambar 3.16.



Gambar 3.16. Contoh pemisahan labeling representasi *level*

Transformasi representasi *current goal* dan *desired goal* untuk DQN + HER dilakukan dengan cara melakukan normalisasi untuk setiap angka pada *current goal* dan *desired goal*. Transformasi representasi *current goal* dan *desired goal* GCSL

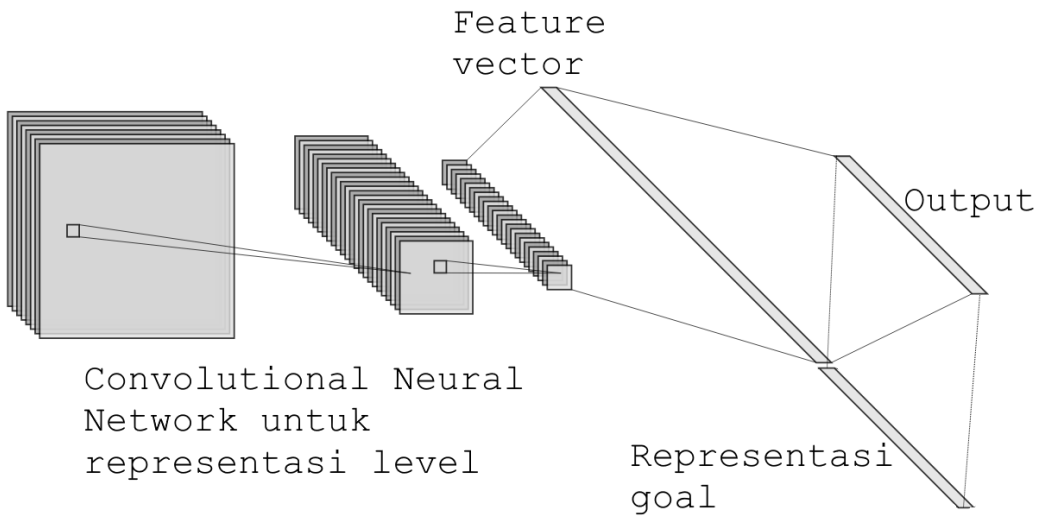
dilakukan dengan cara mengkonversi representasi *tuple* angka menjadi representasi *array* 1 dimensi bernilai 0 atau 1. Kelima angka pada *tuple* masing-masing akan dikonversi menjadi sebuah *array* 1 dimensi bernilai 0 atau 1. Setiap angka akan terletak di awal *array* dan jumlah angka 1 pada *array* sama dengan nilai angka pada *tuple* ditambah 1. Misalnya jumlah musuh 2 (batasan jumlah musuh 0 - 3) direpresentasikan menjadi [1,1,1,0]. Setiap komponen *goal* kemudian akan digabung (menggunakan proses concatenation) menjadi sebuah *vector* 1 dimensi untuk menjadi *input goal*. Sebagai pengingat, *goal* masing-masing terdiri dari 5 angka yang merepresentasikan jarak minimum, jumlah kelelawar, kalajengking, laba-laba, dan *region*.



Gambar 3.17. Contoh pemisahan labeling representasi *goal* untuk *level* berukuran 6 × 6 pada metode DQN + HER dan GCSL

Tiap angka pada representasi *goal* harus diberi batas jumlah yang mungkin untuk menentukan representasi barunya. Jumlah masing-masing jenis musuh

dibatasi sejumlah 0 sampai 3. Batas jarak minimum dan *region* tergantung dari ukuran *level*. Batas bawah dan batas atas ini ditentukan bukan berdasarkan berapa jarak atau jumlah *region* yang boleh dipilih pengguna tetapi berdasarkan berapa jarak atau jumlah *region* yang mungkin terjadi di sebuah *level*. Pada penelitian ini, batas bawah jarak minimum dan jumlah *region* ditentukan menjadi 0. Batas atas jarak minimum ditentukan menjadi $panjang\ level \times lebar\ level \times 2 + 1$ sedangkan batas atas jumlah *region* ditentukan menjadi $panjang\ level \times lebar\ level + 1$. Sebagai contoh, untuk *level* yang berukuran 6×6, representasi jarak minimum dibatasi menjadi 0 sampai 73 dan jumlah *region* pada *level* dibatasi menjadi 0 sampai 37. Contoh pemisahan labeling representasi *goal* dapat dilihat pada Gambar 3.17.



Gambar 3.18. Arsitektur *Neural Network*

Arsitektur *neural network* dari DQN dan GCSL terdiri dari beberapa *layer convolutional neural network* dan dilanjutkan dengan *fully connected network*. *Input* untuk arsitektur *network* ini ada 2, yaitu representasi *level* (*array* 3 dimensi) dan representasi *current goal* + *desired goal* (*array* 1 dimensi). *Convolutional neural network* bertugas untuk memproses *input* representasi *level* yang berbentuk *array* 3 dimensi. *Output* dari *convolutional layer* akhir adalah sebuah vektor 1 dimensi yang berisi informasi fitur laten dari representasi *level*. Vektor *output* ini

kemudian digabung (concatenate) dengan vektor representasi *goal* dari *input*. Gabungan vektor ini kemudian akan diproses menggunakan *fully connected network* untuk menghasilkan *output* dari *neural network*. *Output* dari *neural network* ini kemudian akan digunakan untuk menentukan aksi apa yang akan diambil agen. Gambaran secara garis besar arsitektur *neural network* dapat dilihat pada Gambar 3.18. Ukuran dari masing-masing *layer* tergantung dari ukuran *level* dan akan dipaparkan lebih lanjut pada bagian uji coba.

Output dari *neural network* akan digunakan untuk menentukan aksi yang dikirimkan ke *environment*. Aksi yang dapat dilakukan pada *environment* dikirim dalam format (x, y, t), yaitu mengubah isi petak koordinat (y, x) dengan *tile* t. Oleh karena itu, *output* dari *neural network* DQN dan GCSL adalah sebuah vektor 1 dimensi berisi bilangan real dengan ukuran $(panjang\ level \times lebar\ level \times banyak\ jenis\ petak)$. Sebagai contoh, untuk *level* berukuran 6×6 dengan 8 jenis petak yang berbeda, *output* dari *neural network* adalah vektor 1 dimensi yang berisi 288 bilangan real.

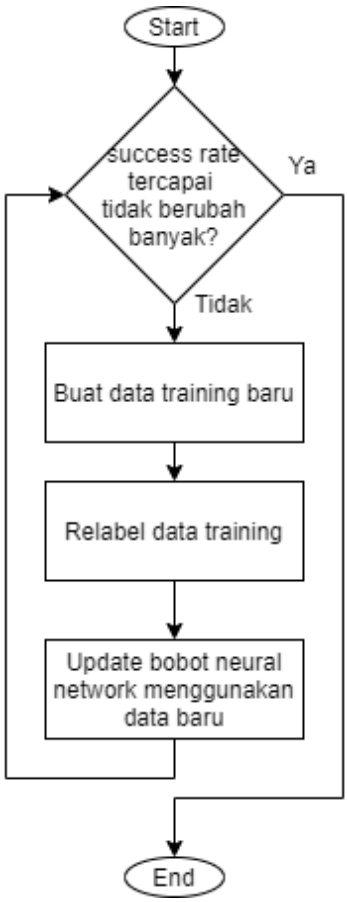
Nilai dari masing-masing angka dalam vektor *output* ini memiliki arti yang berbeda. Untuk arsitektur DQN, nilai masing-masing elemen menyatakan nilai dari Q-*Function* dari masing-masing aksi yang dapat dipilih. Untuk arsitektur GCSL, nilai masing-masing elemen menyatakan tingkat keyakinan *neural network* untuk memilih aksi tersebut. Untuk kedua *neural network*, aksi yang akan dipilih pada tahap testing adalah aksi yang memiliki nilai tertinggi (digunakan fungsi argmax). Untuk proses *training*, metode memilih aksi yang akan dilakukan akan dijelaskan pada subbab pelatihan *neural network*.

Setelah aksi dipilih, akan dilakukan pengubahan representasi aksi yang dipilih menjadi representasi yang dapat diproses oleh *environment*. *Output* dari *neural network* adalah sebuah bilangan bulat (hasil dari operasi argmax) sedangkan aksi yang dapat diterima oleh *environment* adalah sebuah *tuple* 3 angka (x, y, t). Transformasi *output* dilakukan dengan cara melakukan pemetaan antara *output neural network* dengan aksi yang dapat diterima oleh *environment*. Sebagai contoh, untuk ukuran *level* 6×6, angka 0, 1, ..., 7 dipetakan ke (0, 0, 0), (0, 0, 1), ..., (0, 0, 7). Angka 8, 9, ..., 15 dipetakan ke (0, 1, 0), (0, 1, 1), ..., (0, 1, 7). Angka 40, 41, ...,

47 dipetakan ke (0, 5, 0), (0, 5, 1), ..., (0, 5, 7). Angka 48, 49, ..., 55 dipetakan ke (1, 0, 0), (1, 0, 1), ..., (1, 0, 7) dan seterusnya hingga angka 287.

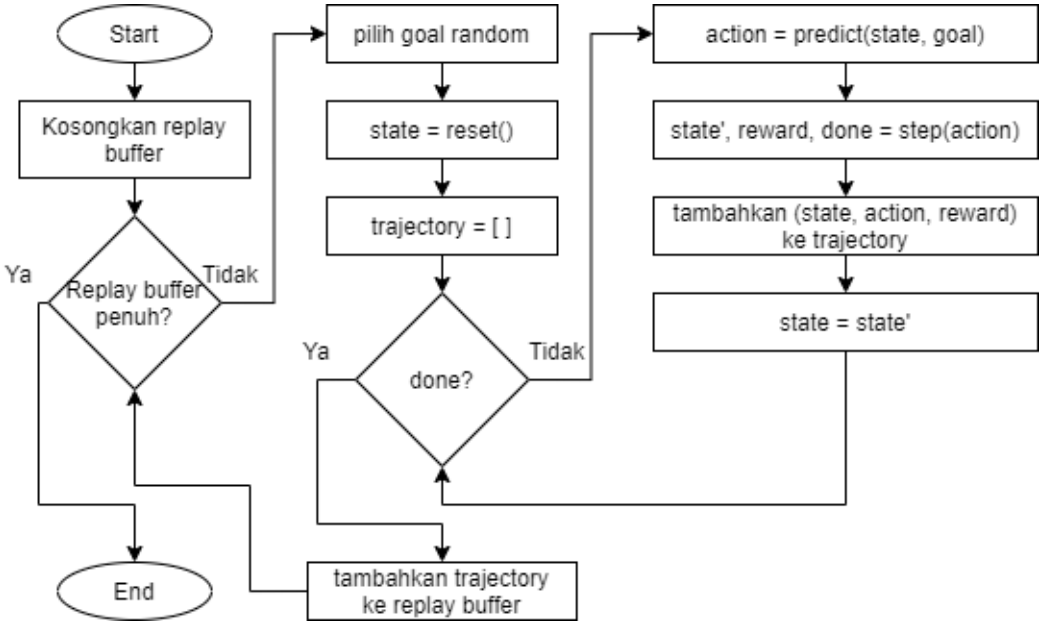
3.2.4 Pelatihan *neural network*

Proses pelatihan *level generator* dengan metode *reinforcement learning* dapat dibagi menjadi 3 bagian: mengumpulkan data *training*, proses *relabeling* untuk data *training*, dan proses *update* bobot *neural network*. Alur proses pelatihan *level generator* secara garis besar digambarkan pada Gambar 3.19. Proses ini dijalankan secara terus menerus hingga batas yang ditentukan. Pada penelitian ini proses *training* dijalankan hingga dirasa tidak ada perubahan signifikan pada *success rate* dari *level generator* selama periode waktu tertentu. Metode pelatihan ini serupa dengan metode pelatihan yang dilakukan pada (Andrychowicz *et al.*, 2017; Hill *et al.*, 2018; Ghosh *et al.*, 2019)



Gambar 3.19. Diagram alir proses pelatihan *level generator*

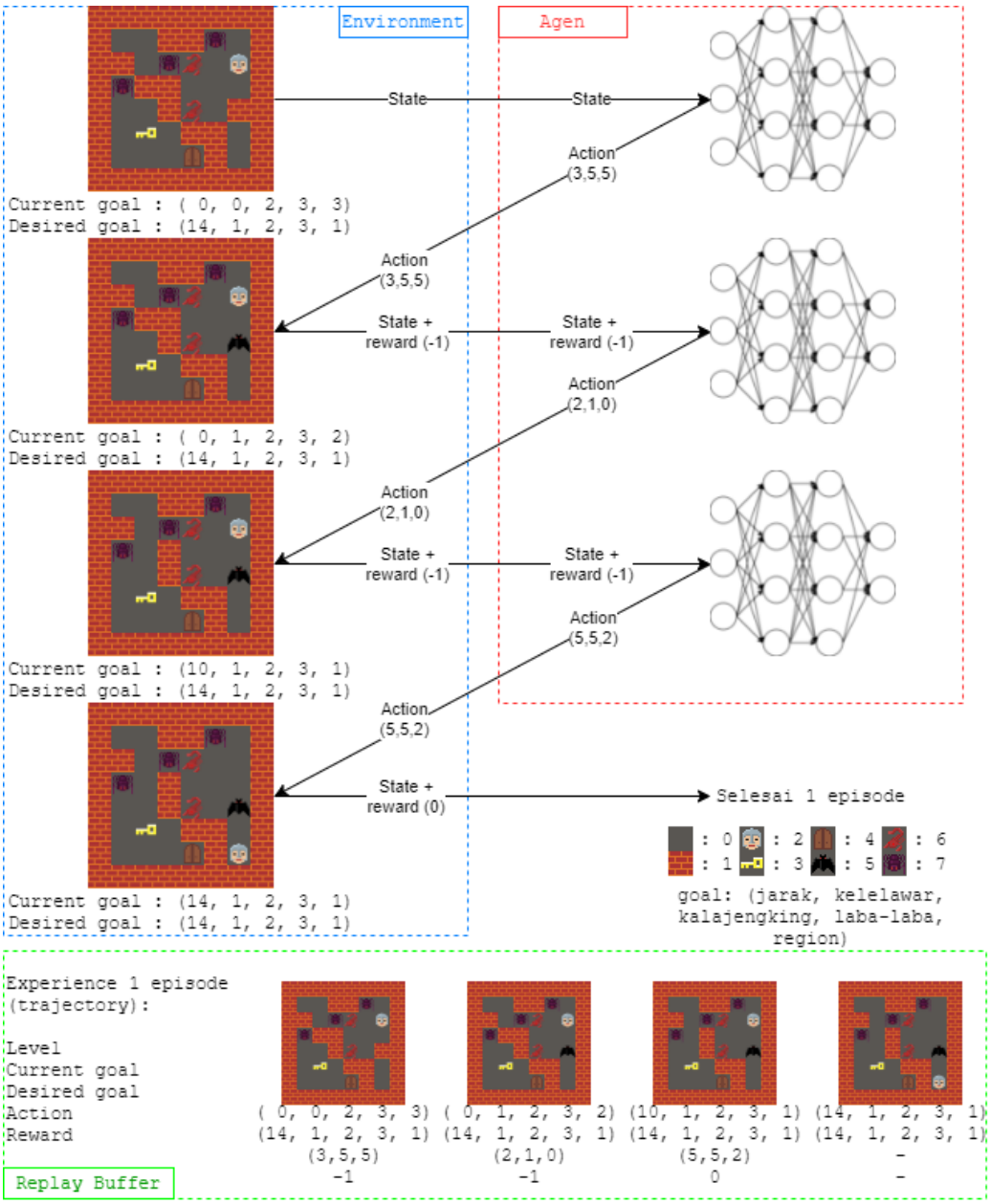
Proses pengumpulan data *training* dapat dilihat pada diagram alir pada Gambar 3.20. Pertama-tama, *environment* diberi perintah *reset* untuk memulai sebuah episode dan mendapatkan *state* awal. Setelah itu, *neural network* akan memproses *state* yang diberikan untuk menghasilkan *output* berupa aksi apa yang harus diambil. Aksi ini kemudian akan diproses oleh *environment* untuk menentukan *state* berikutnya dan *reward* yang diperoleh. *State*, aksi, dan *reward* pada tiap langkah akan disimpan dalam sebuah variabel. Tiga langkah terakhir akan diulangi terus menerus hingga episode berakhir. Setelah episode berakhir, setiap pasang *state*, aksi, dan *reward* akan disimpan pada *replay buffer*. Ilustrasi proses pengumpulan data *training* dapat dilihat pada Gambar 3.21.



Gambar 3.20. Diagram alir proses pengumpulan data *training*

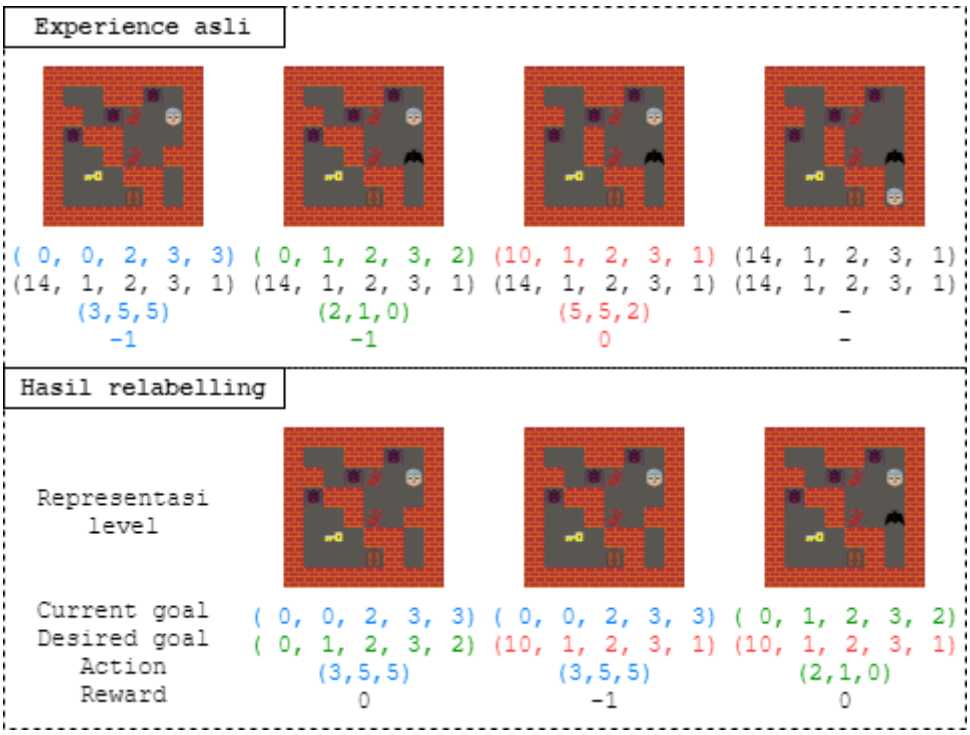
Proses pemilihan aksi yang harus diambil pada fase pelatihan agak berbeda diantara metode DQN dan GCSL. Pemilihan aksi yang harus diambil pada DQN menggunakan metode *ϵ-greedy* (Mnih *et al.*, 2013). Pada metode *ϵ-greedy*, terdapat sebuah *parameter* ϵ yang pada awal program bernilai 1. Setiap kali *neural network* hendak memilih aksi, terdapat peluang sebesar ϵ untuk program memilih aksi secara *random* dan peluang sebesar (1-ϵ) untuk program mengambil aksi dengan

nilai Q *function* (hasil perhitungan *neural network*) terbesar. Selama proses *training*, nilai epsilon akan diturunkan secara bertahap hingga mencapai nilai yang ditentukan (biasanya 0.02). Pemilihan aksi yang harus diambil pada GCSL dilakukan secara acak menggunakan bobot (Ghosh *et al.*, 2019). Bobot untuk masing-masing aksi diperoleh dari nilai softmax *output neural network*. Metode pemilihan aksi ini diperlukan agar agen melakukan eksplorasi cara kerja *environment*.



Gambar 3.21. Ilustrasi pengumpulan data *training* untuk 1 episode

Proses *relabeling* untuk DQN dan GCSL dilakukan sesuai dengan algoritma yang telah dipaparkan pada Bab II. Pada *relabeling* DQN + HER, untuk setiap data transisi, akan disimpan (4 + 1) buah transisi pada *replay buffer*. Empat buah transisi awal berupa transisi yang *desired* goalnya telah diganti dan *reward* dikalkulasi ulang, sedangkan 1 transisi yang lain adalah transisi asli. Proses *relabeling* ini disebut dengan proses *relabeling* metode *future* pada (Andrychowicz *et al.*, 2017).



Gambar 3.22. Ilustrasi proses *relabeling*

Pada GCSL, proses *relabeling* tidak dilakukan pada saat penyimpanan transisi tetapi pada saat hendak dilakukan *update* bobot. Hal ini dikarenakan pada GCSL apabila dalam sebuah episode terdapat T buah transisi maka jumlah transisi yang perlu disimpan adalah $\binom{T}{2}$. Metode *relabeling* ini sama dengan yang diimplementasikan pada (Ghosh *et al.*, 2019). Contoh proses telah dijelaskan pada Bab 2 masing-masing di bagian HER dan GCSL. Ilustrasi proses *relabeling* dapat dilihat pada Gambar 3.22. pada proses relabelling, dibuat *experience* baru dengan

cara mengganti *desired goal* dengan *current goal* dari iterasi setelahnya dan *reward* dihitung ulang sesuai dengan *desired goal* baru. Detail *relabeling* dapat dilihat di Bab 2 dan di (Andrychowicz *et al.*, 2017; Ghosh *et al.*, 2019)

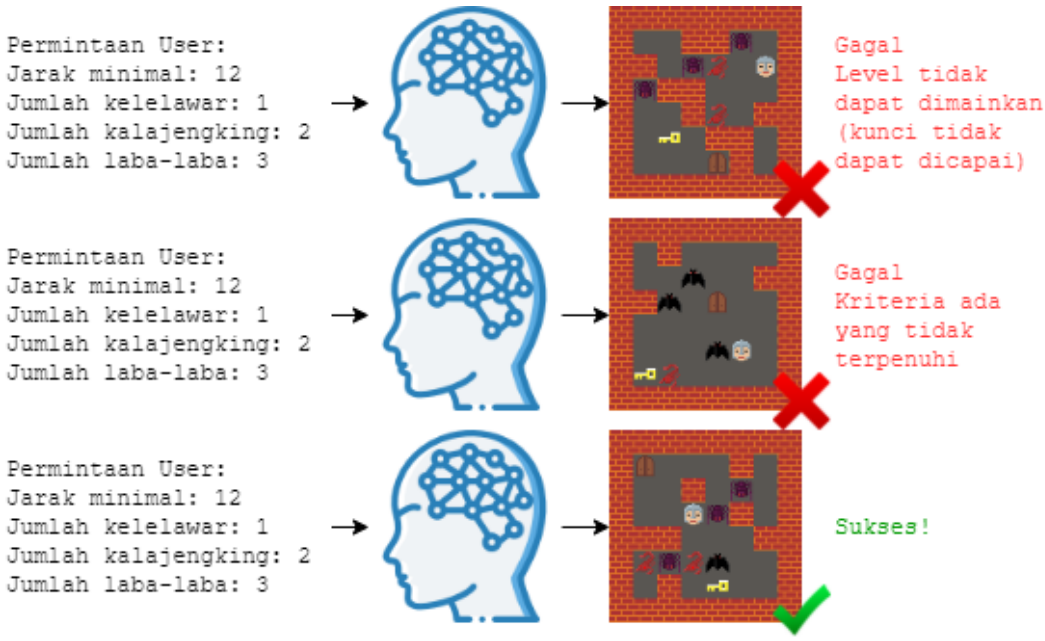
Penyesuaian bobot *neural network* untuk DQN dan GCSL menggunakan metode propagasi balik (backpropagation). *Neural network* untuk DQN dilatih menggunakan teknik pelatihan regresi karena *output* dari *network* ini adalah estimasi nilai Q *function* masing-masing aksi yang dapat dilakukan (Mnih *et al.*, 2013). *Loss function* yang digunakan adalah mean *squared error* (MSE) *loss*. Arsitektur *neural network* dan proses *update* bobot untuk DQN + HER diimplementasikan menggunakan *library stable baselines* dan tensorflow (Hill *et al.*, 2018). *Neural network* untuk GCSL dilatih menggunakan teknik pelatihan klasifikasi karena *output* dari *network* ini adalah aksi mana yang paling baik untuk dilakukan pada *state* saat ini (Ghosh *et al.*, 2019). *Loss function* yang digunakan adalah cross *entropy loss*. Arsitektur *neural network* untuk GCSL diimplementasikan sendiri menggunakan *library* pytorch.

3. 3. Pengujian

Uji coba dilakukan dengan cara mengukur kemampuan agen untuk membuat *level game* yang dapat dimainkan dan memenuhi semua kriteria yang diminta oleh pengguna. *Input* dan *output* dari model untuk proses uji coba sama dengan pada proses pelatihan. *Input* untuk agen adalah kriteria yang diminta oleh pengguna yang harus dipenuhi oleh *level*. *Output* dari agen adalah sebuah *level* hasil modifikasi. Contoh ilustrasi *input* dan *output* untuk uji coba dapat dilihat pada Gambar 3.23.

Agen dianggap sukses membuat *level* apabila *level* yang dihasilkan oleh agen dapat dimainkan dan memenuhi kriteria dari pengguna. *Environment* akan membantu melakukan pengecekan apakah *level* yang dihasilkan dapat dimainkan atau tidak. Pengecekan dilakukan dengan menggunakan algoritma Dijkstra untuk menentukan apakah karakter dapat mencapai kunci dan pintu. *Environment* juga akan membantu menentukan apakah *level* yang dihasilkan sudah memenuhi kriteria yang diminta oleh pengguna dengan cara menghitung jarak minimal, jumlah masing-masing musuh pada *level*, dan jumlah *region* pada *level*. Terdapat

sedikit perbedaan antara metode pemilihan aksi agen pada fase pelatihan dan fase uji coba. Pada fase pelatihan, agen memilih aksi berdasarkan metode yang ditentukan. Metode ini telah dijelaskan pada subbab pelatihan *neural network* (metode ϵ-*greedy* untuk DQN + HER dan *random* berbobot untuk GCSL). Pada fase uji coba, agen memilih aksi yang paling baik berdasarkan interpretasi *output* dari *neural network*. Hal ini berarti secara teori, tingkat kesuksesan agen pada saat uji coba akan lebih tinggi dibanding pada saat fase pelatihan.



Gambar 3.23. Diagram alir penggunaan *environment*

Agen diminta untuk membuat 100 *level* untuk masing-masing kriteria yang bisa dipilih. Sebagai contoh, akan digunakan *level* berukuran 6×6. Untuk *level* berukuran 6×6, terdapat 3 jenis musuh, masing-masing boleh berjumlah 0-3. Selain itu, jarak minimal yang boleh dipilih bernilai 2-16. Jumlah region yang dapat dipilih hanya 1. Dari sini dapat dihitung bahwa total banyak kombinasi kriteria yang dapat dipilih oleh pengguna adalah $4 \times 4 \times 4 \times 15 \times 1 = 960$. Contoh beberapa kriteria yang dapat dipilih adalah $(10, 1, 0, 3, 1)$, yaitu jarak minimal 10, kelelawar 1, kalajengking tidak ada, laba-laba 3, dan region 1; $(15, 2, 1, 3, 1)$; dan $(2, 0, 0, 1, 1)$.

Performansi dari agen RL akan dibandingkan dengan agen yang bekerja secara *random* untuk menunjukkan bahwa agen yang dilatih mampu membuat *level* dengan kriteria yang diberikan. Performansi agen akan diukur berdasarkan berapa persen *level* yang dihasilkan yang memenuhi kriteria yang diberikan. Metrik pengukuran tingkat kesuksesan agen dalam memenuhi kriteria pengguna digunakan pada penelitian-penelitian sebelumnya (Khalifa et al., 2020;Andrychowicz et al., 2017;Ghosh et al., 2019). Penelitian ini dianggap berhasil jika performansi dari agen minimal dua kali lebih baik daripada agen yang bekerja secara *random*.

BAB 4
HASIL DAN PEMBAHASAN

Pada pembahasan ini diberikan pemaparan mengenai implementasi sistem serta pengujian dari sistem berdasarkan skenario yang telah dirancang pada Bab 3. Proses implementasi dilakukan berdasarkan tahapan yang telah diberikan pada pembahasan sebelumnya. Selanjutnya pengujian sistem dilakukan dengan beberapa kondisi yang disesuaikan dengan skenario pengujian. Dari hasil pengujian yang telah didapatkan, selanjutnya diberikan pembahasan dan analisa dari setiap pengujian yang dilakukan dengan tujuan untuk mendapatkan hasil dari penelitian, sehingga mendapatkan kesimpulan yang diberikan pada pembahasan selanjutnya.

4.1. Lingkungan Uji Coba

Adapun lingkungan perangkat lunak yang digunakan pada uji coba ini adalah sebagai berikut:

a. Sistem operasi Ubuntu 20.04.2 LTS
b. Aplikasi Python 3.6

Lingkungan perangkat keras yang digunakan adalah *computer personal* (*stand-alone*) yang memiliki spesifikasi sebagai berikut:

a. *Processor* : Intel Core i7-8700K CPU @ 3.70GHz
b. *Installed memory* : 32 GB
c. *System type* : 64-bit *Operating System*
d. *GPU* : Nvidia GeForce RTX 2060 SUPER

4.2. Hasil Pengujian dan Analisis

Pada subbab ini akan dijelaskan tentang hasil uji coba untuk masing-masing arsitektur. Terdapat 2 arsitektur berbeda yang digunakan pada uji coba ini, yaitu arsitektur DQN + HER dan arsitektur GCSL. Pada tiap subbab akan dijelaskan tentang detail ukuran arsitektur neural network yang digunakan serta dijelaskan pula tentang tingkat kesuksesan masing-masing arsitektur dalam menghasilkan level sesuai permintaan user.

Dalam penelitian ini, performansi dari agen akan diukur dengan menggunakan success rate. Success rate didefinisikan sebagai banyaknya level yang berhasil dibuat sesuai dengan permintaan pengguna dibagi dengan banyaknya level yang dibuat. Success rate dirumuskan dalam persamaan 4.1. Nilai succcess rate terletak pada rentang 0 – 1. Semakin tinggi nilai success rate maka semakin baik performansi agen. Jika agen mencapai success rate 1 maka agen dianggap bekerja dengan sempurna.

$$Success\ rate = \frac{banyaknya\ level\ yang\ memenuhi\ kriteria}{banyaknya\ level\ yang\ dibuat} \tag{4.1}$$

Pada penelitian ini digunakan 2 buah agen yang berbeda. Agen pertama adalah agen random. Agen random akan melakukan modifikasi secara random pada level. Agen random berguna sebagai tolok ukur performa agen yang telah dilatih. Agen kedua adalah agen yang telah dilatih dengan metode DQN + HER atau GCSL. Agen kedua akan disebut sebagai agen RL pada bab ini.

Tabel 4.1. Arsitektur *neural network* untuk DQN

No	**Jenis *layer***	***Input***	**Keterangan**	***Output***
1	*Convolutional*	$6 \times 6 \times 8$	Kernel: $3 \times 3 \times 32$ Stride: 1 Padding: 1	$6 \times 6 \times 32$
3	ReLU	$6 \times 6 \times 32$		$6 \times 6 \times 32$
4	*Convolutional*	$6 \times 6 \times 32$	Kernel: $3 \times 3 \times 64$ Stride: 1 Padding: 1	$6 \times 6 \times 64$
6	ReLU	$6 \times 6 \times 64$		$6 \times 6 \times 64$
7	*Convolutional*	$6 \times 6 \times 64$	Kernel: $3 \times 3 \times 64$ Stride: 1 Padding: 1	$6 \times 6 \times 64$
9	ReLU	$6 \times 6 \times 64$		$6 \times 6 \times 64$
10	Flatten	$6 \times 6 \times 64$		4608

No	Jenis *layer*	*Input*	Keterangan	*Output*
11	Concatenate	$2304 + 5 \times 2$		2314
12	Dense	2314		1152
14	ReLU	1152		1152
16	Dense	1152		576
18	ReLU	576		576
19	Dense	576		288

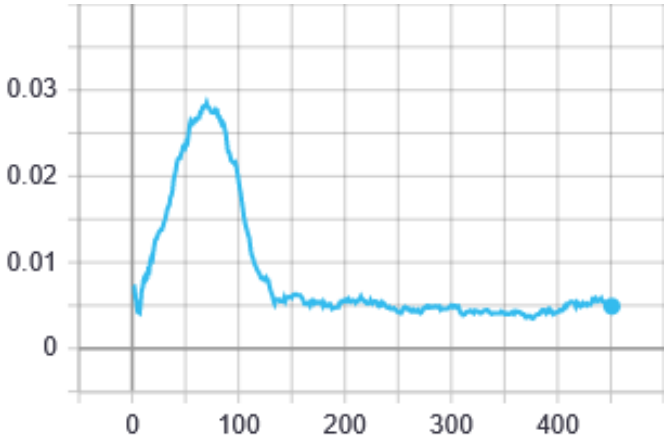
4.2.1 Pengujian Arsitektur DQN + HER

Sebelum dilakukan pengujian, pertama-tama dijalankan proses pelatihan *neural network* terlebih dahulu. Arsitektur *neural network* dan proses *update* bobot untuk DQN + HER diimplementasikan menggunakan *library stable baselines* dan tensorflow. Pada pelatihan pertama ini digunakan *environment game* Zelda dengan ukuran 6×6. Arsitektur *neural network* dan *hyperparameter* yang digunakan untuk DQN + HER dapat dilihat pada Tabel 4.1 dan 4.2. Perkembangan nilai *success rate* selama berjalannya proses *training* dapat dilihat pada Gambar 4.1.

Tabel 4.2. *Hyperparameter* yang digunakan untuk pelatihan DQN

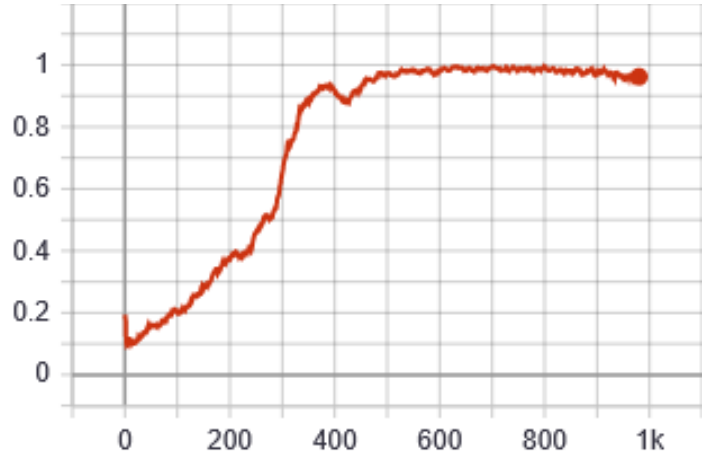
Hyperparameter	**Nilai**
***Learning rate* (α)**	0.05
Batch *size*	256
Buffer size	1000000
Learning starts	5000
Target *network update frequency*	10000

Dari perkembangan *success rate* selama proses *training* dapat dilihat bahwa arsitektur DQN + HER tidak mampu menghasilkan sebuah *level generator* yang baik. Dapat dilihat bahwa *success rate* sempat naik di awal *training*, namun kemudian turun dan tidak bisa naik kembali. Nilai *success rate* maksimal yang dapat dicapai adalah 0.03.



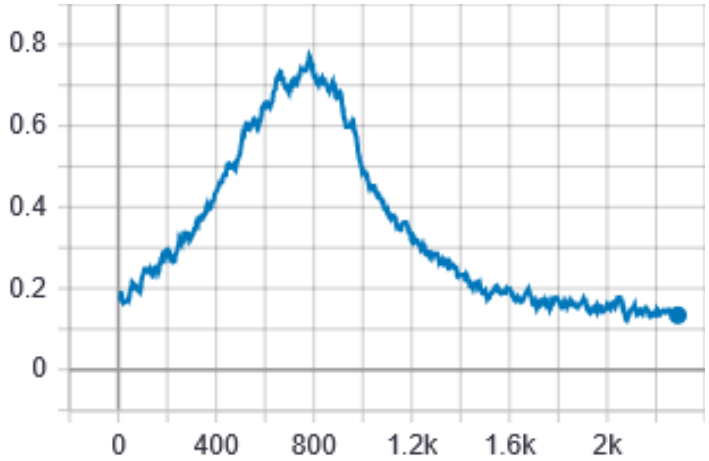
Gambar 4.1. Perkembangan success rate selama proses training DQN + HER untuk environment Zelda berukuran 6×6

Untuk mencari penyebab kegagalan pelatihan arsitektur ini, dibuat 2 buah *environment* sederhana (toy *environment*), yaitu *environment* wall dan *region*. Kedua *environment* ini hanya memiliki 2 jenis petak, yaitu *empty* (kosong) dan tembok (*solid*). Kriteria yang dapat dipilih untuk *environment* wall adalah berapa banyak petak tembok yang harus ada pada *level* sedangkan untuk *environment region* kriteria yang dapat dipilih adalah banyaknya *region* yang diinginkan pada *level*. Perkembangan nilai *success rate* selama berjalannya proses *training* untuk *environment region* dan wall dapat dilihat pada Gambar 4.2 dan 4.3. Selain kedua toy *environment* ini, dibuat juga *environment* sederhana dengan representasi *level* berupa *array* 1 dimensi. Detail lebih lanjut untuk semua *environment* ini dilampirkan pada lampiran 1.



Gambar 4.2. Perkembangan success rate selama proses training DQN + HER untuk toy environment regions berukuran 6×6

Dari perkembangan *success rate* selama proses *training* dapat dilihat bahwa arsitektur DQN + HER dapat menghasilkan *level* untuk toy *environment*. Untuk *environment region*, *level generator* yang dihasilkan memiliki *success rate* maksimum mendekati 1 dan untuk *environment* wall, nilai *success rate* maksimum *level generator* mendekati 0.8. Namun pada *environment* wall, terjadi permasalahan yang sama dengan pada proses pelatihan zelda, yaitu nilai *success rate* yang menurun di tengah-tengah pelatihan dan tidak dapat kembali naik. Karena ketidakstabilan pelatihan metode DQN, maka diputuskan untuk menghentikan uji coba dengan menggunakan DQN dan dilanjutkan dengan pengunaan metode baru (GCSL).



Gambar 4.3. Perkembangan success rate selama proses training DQN + HER untuk toy environment wall berukuran 6×6

4.2.2 Pengujian Arsitektur GCSL

Sebelum dilakukan pengujian, pertama-tama dijalankan proses pelatihan *neural network* terlebih dahulu. Arsitektur *neural network* dan proses *update* bobot untuk GCSL diimplementasikan sendiri menggunakan *library* Pytorch. Pada pelatihan pertama ini digunakan *environment game* Zelda dengan ukuran 6×6. *Hyperparameter* dan arsitektur *neural network* yang digunakan untuk GCSL dapat dilihat pada Tabel 4.3 dan 4.4. Perkembangan nilai *success rate* selama berjalannya proses *training* dapat dilihat pada Gambar 4.4.

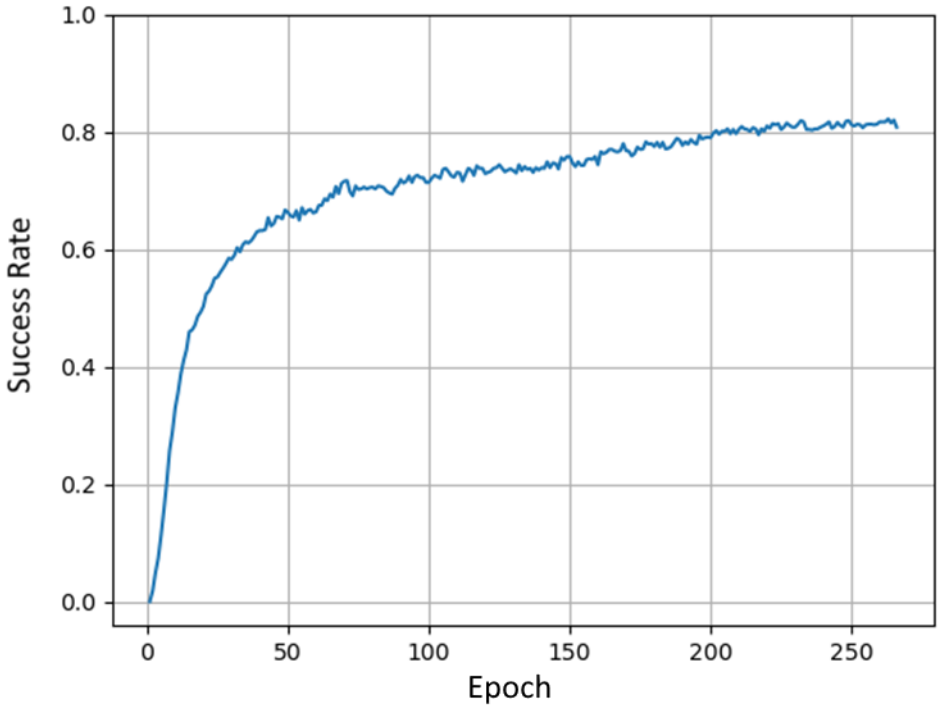
Tabel 4.3. Arsitektur *neural network* GCSL untuk *environment* Zelda berukuran **6×6**

No	**Jenis *layer***	***Input***	**Keterangan**	***Output***
1	*Convolutional*	$6 \times 6 \times 8$	Kernel: $3 \times 3 \times 128$ Stride: 1 Padding: 1	$6 \times 6 \times 128$
2	Batch Norm	$6 \times 6 \times 128$		$6 \times 6 \times 128$
3	ReLU	$6 \times 6 \times 128$		$6 \times 6 \times 128$
4	*Convolutional*	$6 \times 6 \times 128$	Kernel: $3 \times 3 \times 128$ Stride: 1 Padding: 1	$6 \times 6 \times 128$
5	Batch Norm	$6 \times 6 \times 128$		$6 \times 6 \times 128$
6	ReLU	$6 \times 6 \times 128$		$6 \times 6 \times 128$
7	*Convolutional*	$6 \times 6 \times 128$	Kernel: $3 \times 3 \times 128$ Stride: 1 Padding: 1	$6 \times 6 \times 128$
8	Batch Norm	$6 \times 6 \times 128$		$6 \times 6 \times 128$
9	ReLU	$6 \times 6 \times 128$		$6 \times 6 \times 128$
10	Flatten	$6 \times 6 \times 128$		4608
11	Concatenate	$4608 + 250 \times 2$		5108
12	Dense	5108		10216
13	Batch Norm	10216		10216
14	ReLU	10216		10216
15	Dropout	10216	Drop: 0.4	10216
16	Dense	10216		10216
17	Batch Norm	10216		10216
18	ReLU	10216		10216
19	Dense	10216		288

Tabel 4.4. *Hyperparameter* yang digunakan untuk pelatihan GCSL

Hyperparameter	**Nilai**
***Discount* factor (γ)**	0.99
***Learning rate* (α)**	0.0005
Batch *size*	256
Buffer size	50000

Dari perkembangan *success rate* selama proses *training* dapat dilihat bahwa arsitektur GCSL mampu menghasilkan sebuah *level generator* yang baik. Nilai *success rate* maksimal yang dapat dicapai adalah 0.822. Dapat dilihat bahwa nilai *success rate* masih bergerak naik namun perubahannya tidak signifikan sehingga proses *training* dihentikan setelah 266 epoch.

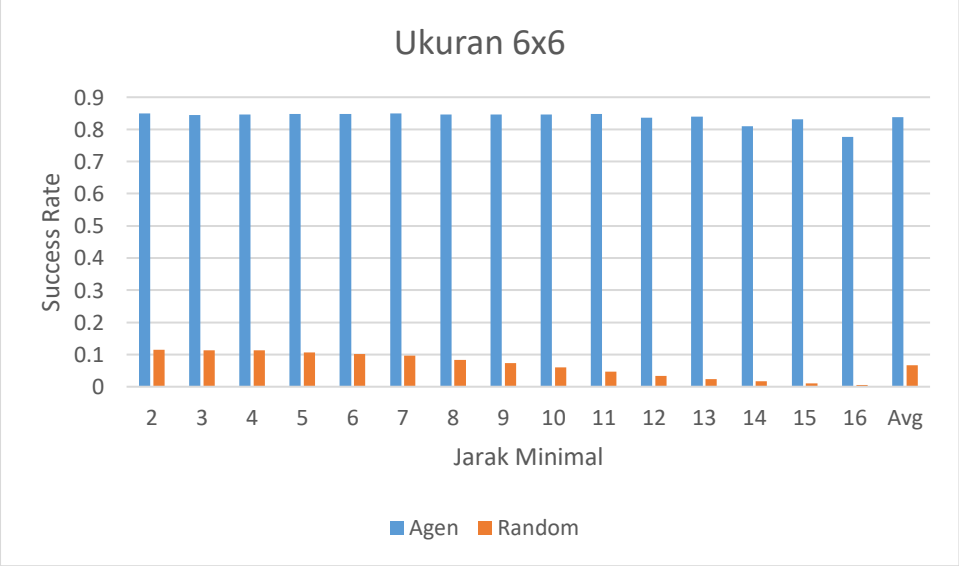


Gambar 4.4. Perkembangan success rate selama proses training GCSL untuk environment Zelda berukuran 6×6

Setelah proses pelatihan, dilakukan proses uji coba performa *level generator*. Pengujian arsitektur GCSL dilakukan dengan cara membuat 100 buah *level* untuk

masing-masing kriteria yang dapat diberikan. Untuk *level* berukuran 6×6, terdapat 3 jenis musuh, masing-masing berjumlah 0-3, dan jarak minimal yang boleh dipilih bernilai 2-16 menghasilkan banyak kombinasi $4 \times 4 \times 4 \times 15 = 960$ kriteria yang dapat dipilih. Untuk mempermudah penyajian data hasil, data *success rate* yang ditampilkan akan dikelompokkan berdasarkan jarak minimum. Nilai *success rate* agen dikelompokkan berdasar jarak minimum dapat dilihat pada Gambar 4.5. Tabel *success rate* lengkap untuk masing-masing kriteria yang dapat dipilih oleh pengguna dilampirkan pada lampiran 2. Contoh *level* yang berhasil dibuat oleh agen RL dapat dilihat pada Tabel 4.6. Contoh proses pembuatan sebuah *level* dapat dilihat pada lampiran 3.

Pada Gambar 4.5. dapat dilihat bahwa *success rate* dari agen yang dibuat dengan menggunakan algoritma GCSL mencapai nilai rata-rata 0.838. Nilai ini jauh lebih tinggi apabila dibandingkan dengan dengan agen yang berjalan secara *random*, yaitu 0.067. Dapat dilihat juga bahwa semakin besar jarak minimum yang diminta, terjadi sedikit penurunan *success rate* dari kedua agen.

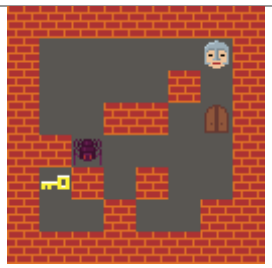
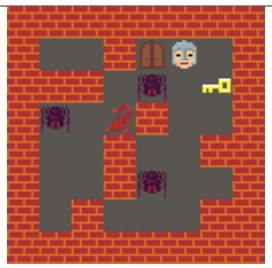


Gambar 4.5. Perbandingan *success rate* agen *random* dan agen yang dilatih menggunakan GCSL untuk *environment* Zelda berukuran $\mathbf{6 \times 6}$, dikelompokkan berdasar jarak minimal

Pelatihan arsitektur berikutnya dilakukan untuk *environment* berukuran 7×9. Arsitektur *neural network* dan proses *update* bobot untuk GCSL

diimplementasikan sendiri menggunakan *library*. Arsitektur *neural network* yang digunakan untuk GCSL dapat dilihat pada Tabel 4.7. *Hyperparameter* yang digunakan sama dengan *environment* 6×6 dan dapat dilihat pada Tabel 4.4. Perkembangan nilai *success rate* selama berjalannya proses *training* dapat dilihat pada Gambar 4.6.

Tabel 4.5. Contoh *level* yang berhasil dibuat oleh agen RL untuk *environment* Zelda berukuran **6 × 6**

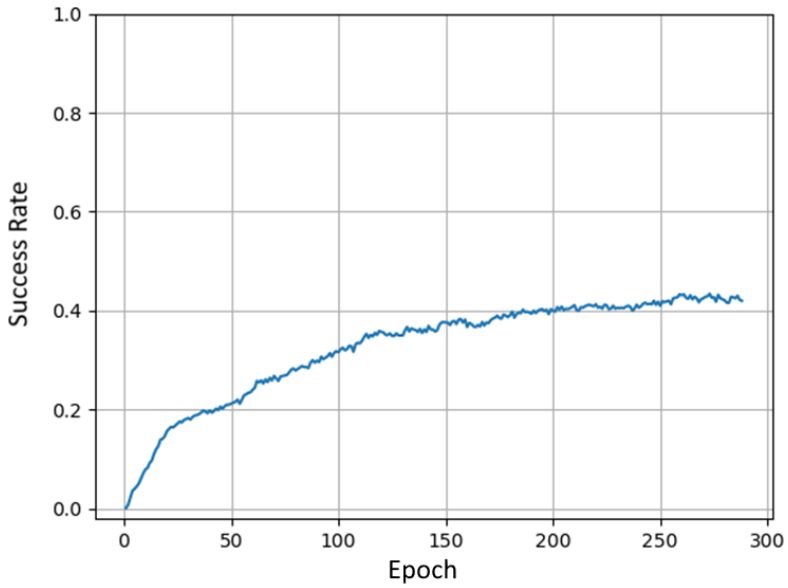
No	***Desired Goal***	**Initial *State***	**Result**
1	(16, 1, 2, 0, 1)		
2	(16, 3, 1, 3, 1)		
3	(15, 2, 2, 3, 1)		

Dari perkembangan *success rate* selama proses *training* dapat dilihat bahwa untuk ukuran *environment* 79, *success rate* yang dicapai sekitar setengah dari skenario sebelumnya. Nilai *success rate* maksimal yang dapat dicapai adalah 0.434. Dapat dilihat bahwa nilai *success rate* masih bergerak naik namun perubahannya tidak signifikan sehingga proses *training* dihentikan setelah 288 epoch. Walaupun

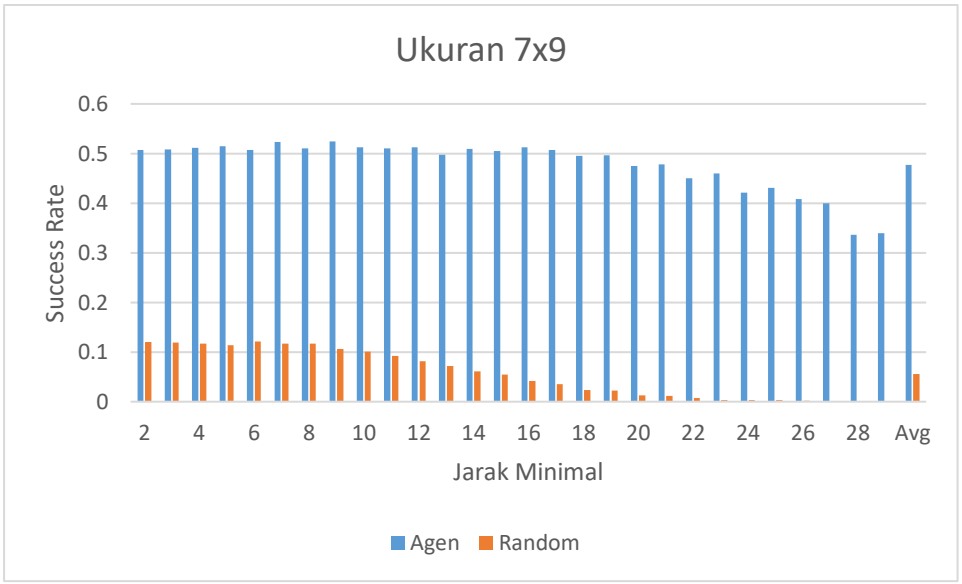
tidak sebaik performansi sebelumnya, nilai ini masih jauh lebih tinggi jika dibandingkan *generator* yang bekerja secara *random*.

Tabel 4.6. Arsitektur *neural network* GCSL untuk *environment* Zelda berukuran $\mathbf{7 \times 9}$

No	**Jenis *layer***	***Input***	**Keterangan**	***Output***
1	*Convolutional*	$7 \times 9 \times 8$	Kernel: $3 \times 3 \times 128$ Stride: 1 Padding: 1	$7 \times 9 \times 128$
2	Batch Norm	$7 \times 9 \times 128$		$7 \times 9 \times 128$
3	ReLU	$7 \times 9 \times 128$		$7 \times 9 \times 128$
4	*Convolutional*	$7 \times 9 \times 128$	Kernel: $3 \times 3 \times 128$ Stride: 1 Padding: 1	$7 \times 9 \times 128$
5	Batch Norm	$7 \times 9 \times 128$		$7 \times 9 \times 128$
6	ReLU	$7 \times 9 \times 128$		$7 \times 9 \times 128$
7	*Convolutional*	$7 \times 9 \times 128$	Kernel: $3 \times 3 \times 128$ Stride: 1 Padding: 1	$7 \times 9 \times 128$
8	Batch Norm	$7 \times 9 \times 128$		$7 \times 9 \times 128$
9	ReLU	$7 \times 9 \times 128$		$7 \times 9 \times 128$
10	Flatten	$7 \times 9 \times 128$		8064
11	Concatenate	$8064 + 331 \times 2$		8726
12	Dense	8726		17452
13	Batch Norm	17452		17452
14	ReLU	17452		17452
15	Dropout	17452	Drop: 0.4	17452
16	Dense	17452		17452
17	Batch Norm	17452		17452
18	ReLU	17452		17452
19	Dense	17452		504



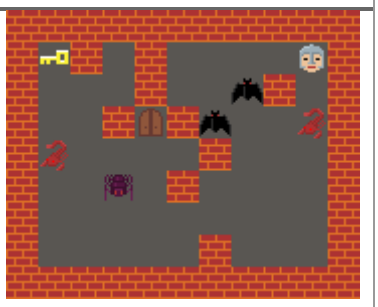
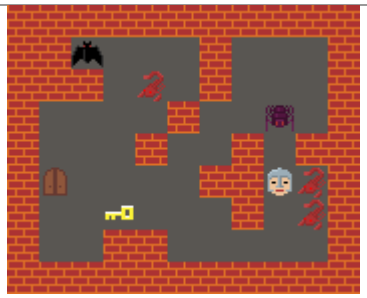
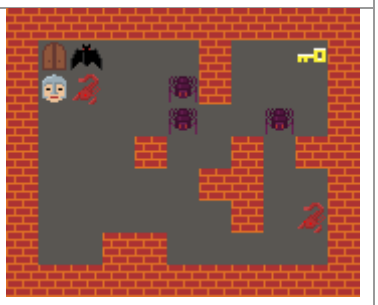
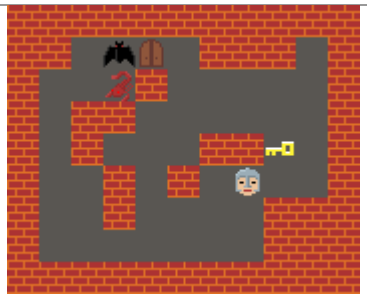
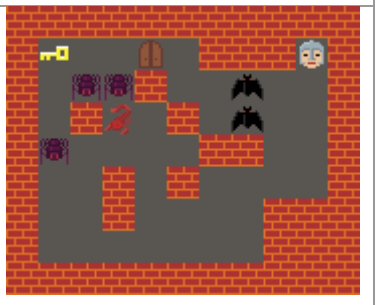
Gambar 4.6. Perkembangan success rate selama proses training GCSL untuk environment Zelda berukuran 7×9



Gambar 4.7. Perbandingan *success rate* agen *random* dan agen yang dilatih menggunakan GCSL untuk *environment* Zelda berukuran **7 × 9**, dikelompokkan berdasar jarak minimal

Setelah proses pelatihan, dilakukan proses uji coba performa *level generator*. Untuk *level* berukuran 7×9, jarak minimal yang boleh dipilih bernilai 2-29 menghasilkan banyak kombinasi $4 \times 4 \times 4 \times 28 = 1792$ kriteria yang dapat dipilih. Nilai *success rate* agen dikelompokkan berdasar jarak minimum dapat dilihat pada Gambar 4.7. Tabel *success rate* lengkap untuk masing-masing kriteria yang dapat dipilih oleh pengguna dilampirkan pada lampiran 2. Contoh *level* yang berhasil dibuat oleh agen RL dapat dilihat pada Tabel 4.9. Contoh proses pembuatan sebuah *level* dapat dilihat pada lampiran 3.

Tabel 4.7. Contoh *level* yang berhasil dibuat oleh agen RL untuk *environment* Zelda berukuran **7 × 9**

No	***Desired Goal***	**Initial *State***	**Result**
1	(17, 2, 2, 1, 1)		
2	(23, 1, 2, 3, 1)		
3	(18, 2, 1, 3, 1)		

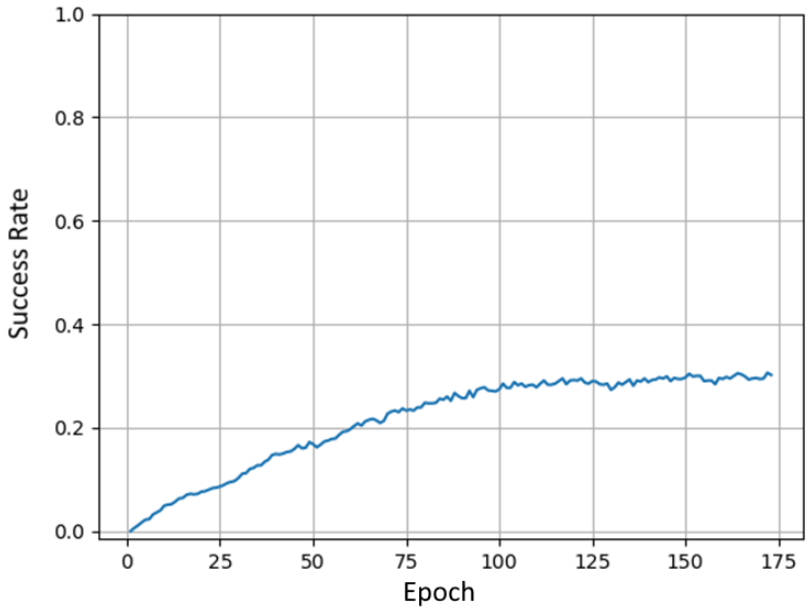
Pada Gambar 4.7. dapat dilihat bahwa *success rate* dari agen yang dibuat dengan menggunakan algoritma GCSL mencapai nilai rata-rata 0.477. Nilai ini tetap jauh lebih tinggi apabila dibandingkan dengan dengan agen yang berjalan secara *random*, yaitu 0.056. Dapat dilihat juga bahwa semakin besar jarak minimum yang diminta, terjadi penurunan *success rate* dari kedua agen. Penurunan ini lebih besar jika dibandingkan dengan ukuran peta 6×6.

Pelatihan arsitektur berikutnya dilakukan untuk *environment* berukuran 7×11. Arsitektur *neural network* dan proses *update* bobot untuk GCSL diimplementasikan sendiri menggunakan *library*. Arsitektur *neural network* yang digunakan untuk GCSL dapat dilihat pada Tabel 4.10. *Hyperparameter* yang digunakan sama dengan *environment* 6×6 dan dapat dilihat pada Tabel 4.4. Perkembangan nilai *success rate* selama berjalannya proses *training* dapat dilihat pada Gambar 4.8.

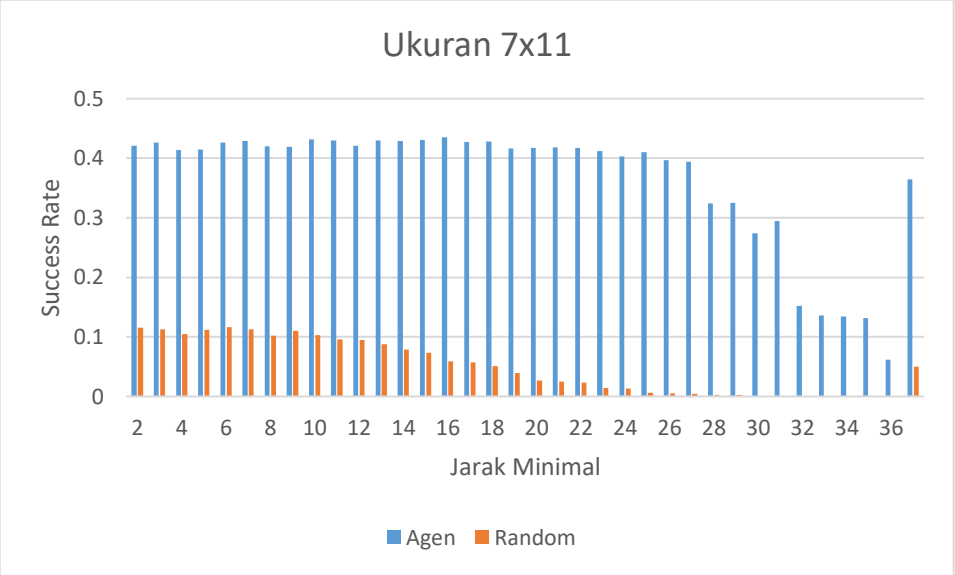
Tabel 4.8. Arsitektur neural network GCSL untuk environment Zelda berukuran **7×11**

No	**Jenis *layer***	***Input***	**Keterangan**	***Output***
1	*Convolutional*	$7 \times 11 \times 8$	Kernel: $3 \times 3 \times 196$ Stride: 1 Padding: 1	$7 \times 11 \times 96$
2	Batch Norm	$7 \times 11 \times 96$		$7 \times 11 \times 96$
3	ReLU	$7 \times 11 \times 96$		$7 \times 11 \times 96$
4	*Convolutional*	$7 \times 11 \times 96$	Kernel: $3 \times 3 \times 196$ Stride: 1 Padding: 1	$7 \times 11 \times 96$
5	Batch Norm	$7 \times 11 \times 96$		$7 \times 11 \times 96$
6	ReLU	$7 \times 11 \times 96$		$7 \times 11 \times 96$
7	*Convolutional*	$7 \times 11 \times 96$	Kernel: $3 \times 3 \times 196$ Stride: 1 Padding: 1	$7 \times 11 \times 96$
8	Batch Norm	$7 \times 11 \times 96$		$7 \times 11 \times 96$

No	Jenis *layer*	*Input*	Keterangan	*Output*
9	ReLU	$7 \times 11 \times 96$		$7 \times 11 \times 96$
10	Flatten	$7 \times 11 \times 96$		7392
11	Concatenate	$7392 + 341 \times 2$		8074
12	Dense	8074		16148
13	Batch Norm	16148		16148
14	ReLU	16148		16148
15	Dropout	16148	Drop: 0.4	16148
16	Dense	16148		16148
17	Batch Norm	16148		16148
18	ReLU	16148		16148
19	Dense	16148		616



Gambar 4.8. Perkembangan success rate selama proses training GCSL untuk environment Zelda berukuran 7×11



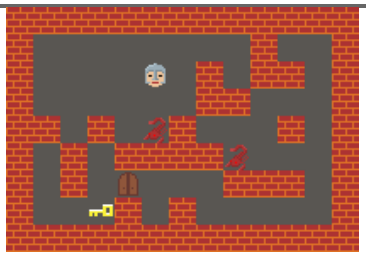
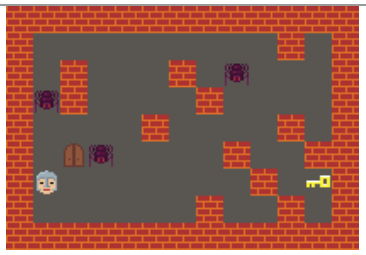
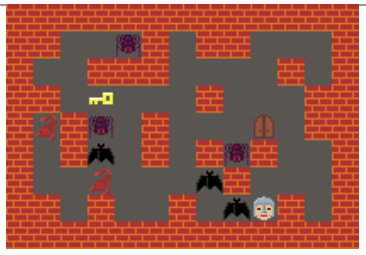
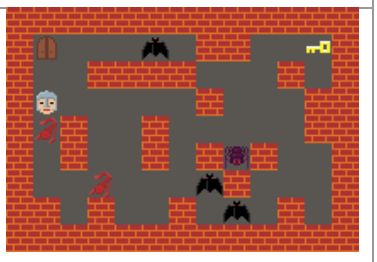
Gambar 4.9. Perbandingan *success rate* agen *random* dan agen yang dilatih menggunakan GCSL untuk *environment* Zelda berukuran 7× **11**, dikelompokkan berdasar jarak minimal

Dari perkembangan *success rate* selama proses *training* dapat dilihat bahwa untuk ukuran *environment* 7×11, *success rate* maksimal kembali menurun. Nilai *success rate* maksimal yang dapat dicapai adalah 0.3. Dapat dilihat bahwa nilai *success rate* masih bergerak naik namun perubahannya tidak signifikan sehingga proses *training* dihentikan setelah 173 epoch. Walaupun tidak sebaik performansi sebelumnya, nilai ini masih jauh lebih tinggi jika dibandingkan *generator* yang bekerja secara *random*.

Setelah proses pelatihan, dilakukan proses uji coba performa *level generator*. Untuk *level* berukuran 7×11, terdapat 3 jenis musuh, masing-masing berjumlah 0-3, dan jarak minimal yang boleh dipilih bernilai 2-35 menghasilkan banyak kombinasi $4 \times 4 \times 4 \times 35 = 2240$ kriteria yang dapat dipilih. Nilai *success rate* agen dikelompokkan berdasar jarak minimum dapat dilihat pada Gambar 4.9. Tabel *success rate* lengkap untuk masing-masing kriteria yang dapat dipilih oleh pengguna dilampirkan pada lampiran 2. Contoh *level* yang berhasil dibuat oleh agen RL dapat dilihat pada Tabel 4.12. Contoh proses pembuatan sebuah *level* dapat dilihat pada lampiran 3.

Pada Gambar 4.9. dapat dilihat bahwa *success rate* dari agen yang dibuat dengan menggunakan algoritma GCSL mencapai nilai rata-rata 0.364. Nilai ini tetap jauh lebih tinggi apabila dibandingkan dengan dengan agen yang berjalan secara *random*, yaitu 0.050. Dapat dilihat juga bahwa semakin besar jarak minimum yang diminta, terjadi penurunan *success rate* secara signifikan dari kedua agen. Penurunan ini lebih besar jika dibandingkan dengan kedua ukuran peta sebelumnya.

Tabel 4.9. Contoh *level* yang berhasil dibuat oleh agen RL untuk *environment* Zelda berukuran **7 × 11**

No	*Desired Goal*	Initial *State*	Result
1	(10, 2, 1, 0, 1)		
2	(22, 2, 3, 3, 1)		
3	(28, 3, 2, 1, 1)		

4.3. Evaluasi Hasil Uji Coba

Berdasarkan pelaksanaan dan hasil uji coba untuk masing-masing arsitektur dan ukuran level game, diperoleh temuan-temuan sebagai berikut:

1. Metode DQN + HER dapat digunakan untuk membuat level generator jika level yang dibuat cukup sederhana. Misalnya membuat maze / labirin yang hanya memiliki 2 jenis tile (tembok dan kosong).
2. Untuk game yang lebih kompleks, DQN kurang stabil (success rate turun dan tidak dapat kembali naik) sehingga sulit untuk membuat level generator dengan success rate yang tinggi.
3. Berdasarkan hasil uji coba, metode GCSL memiliki rata-rata success rate 0.838 untuk level berukuran 6×6, 0.477 untuk level berukuran 7×9, 0.364 untuk level berukuran 7×11. Metode GCSL memiliki rata-rata success rate lebih dari 2 kali success rate agen random, yaitu 0.067 untuk level berukuran 6×6, 0.056 untuk level berukuran 7×9, 0.050 untuk level berukuran 7×11.
4. Terdapat pola bahwa tingkat kesuksesan agen semakin menurun ketika diminta untuk membuat level dengan jarak minimum yang tinggi. Hal ini disebabkan level yang memiliki jarak minimum tinggi jarang ditemui. Hal ini dapat dilihat dari success rate level generator yang berjalan secara random.
5. Terdapat pola bahwa tingkat kesuksesan agen semakin menurun ketika diminta untuk membuat level dengan jumlah musuh yang lebih sedikit. Peyebabnya sama dengan poin sebelumnya, yaitu level dengan jumlah musuh yang lebih sedikit lebih langka ditemui.
6. Dapat dilihat pada Lampiran 3, agen masih beberapa kali memilih langkah modifikasi yang kurang optimum walaupun pada akhirnya berhasil membuat level yang memenuhi kriteria. Hal ini kemungkinan disebabkan karena waktu pelatihan yang kurang lama atau karena arsitektur neural network yang kurang kompleks untuk dapat merepresentasikan level dan goal.

[*Halaman ini sengaja dikosongkan*]

BAB 5
KESIMPULAN DAN SARAN

Pada bab terakhir ini, ditarik beberapa kesimpulan yang didapat dari hasil penelitian dan saran-saran yang dapat digunakan sebagai bahan pertimbangan untuk pengembangan atau riset selanjutnya

5.1 Kesimpulan

Berdasarkan hasil uji coba, pembahasan dan temuan yang sebelumnya sudah disesuaikan dengan tujuan penelitian, maka secara keseluruhan penelitian ini dapat disimpulkan sebagai berikut.

1. Untuk membuat sebuah metode *procedural level generation* yang dapat dikustomisasi, algoritma DQN + HER kurang stabil (*success rate* turun dan tidak dapat kembali naik) dan kurang optimal ketika digabungkan dengan PCGRL. Algoritma GCSL yang lebih stabil dapat digunakan sebagai pengganti DQN+HER untuk digabungkan dengan PCGRL dan dari hasil percobaan, GCSL dapat menghasilkan metode *procedural level generation* dengan *success rate* yang jauh lebih tinggi dibandingkan agen random.
2. Berdasarkan hasil uji coba, metode GCSL memiliki rata-rata *success rate* 0.838 untuk *level* berukuran 6×6, 0.477 untuk *level* berukuran 7×9, 0.364 untuk *level* berukuran 7×11. Metode GCSL memiliki rata-rata *success rate* lebih dari 2 kali *success rate* agen *random*, yaitu 0.067 untuk *level* berukuran 6×6, 0.056 untuk *level* berukuran 7×9, 0.050 untuk *level* berukuran 7×11.
3. Terdapat pola bahwa tingkat kesuksesan agen semakin menurun ketika diminta untuk membuat *level* dengan jarak minimum yang tinggi. Hal ini disebabkan *level* yang memiliki jarak minimum tinggi jarang ditemui. Hal ini dapat dilihat dari *success rate level generator* yang berjalan secara *random*.

5.2 Saran

Berdasarkan hasil yang diperoleh dari penelitian ini, saran yang diperlukan untuk perbaikan system maupun untuk penelitian selanjutnya adalah sebagai berikut.

1. Dapat digunakan metode eksplorasi yang lebih baik sehingga level dengan jarak minimum tinggi lebih banyak ditemukan, misalnya dengan metode go-explore (Ecoffet *et al.*, 2019).
2. Bisa digunakan neural network yang lebih kompleks atau jenis lain yang lebih cocok untuk merepresentasikan level dengan ukuran besar, misalnya dengan menggunakan resnet (He *et al.*, 2016; Silver *et al.*, 2017) atau menggunakan metode ensemble (Zhou, 2021).

DAFTAR PUSTAKA

Andrychowicz, M. *et al.* (2017) 'Hindsight Experience Replay', *Advances in Neural Information Processing Systems*, 2017-Decem(Nips), pp. 5049–5059.

Bellman, R. (1957) 'A Markovian decision process', *Journal of Mathematics and Mechanics*, pp. 679–684.

Ecoffet, A. *et al.* (2019) 'Go-Explore: a new approach for hard-exploration problems', *arXiv preprint arXiv:1901.10995*.

Ghosh, D. *et al.* (2019) 'Learning to Reach Goals via', pp. 1–21. Available at: http://arxiv.org/abs/1912.06088.

He, K. *et al.* (2016) 'Deep residual learning for image recognition', *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. doi: 10.1109/CVPR.2016.90.

Hill, A. *et al.* (2018) *Stable-Baselines*, *GitHub*.

Khalifa, A. *et al.* (2020) 'PCGRL: Procedural Content Reinforcement Learning'. Available at:

Liu, J. *et al.* (2020) 'Deep learning for procedural content generation', *Neural Computing and Applications*, 8. doi: 10.1007/s00521-020-05383-8.

Mnih, V. *et al.* (2013) 'Playing atari with deep reinforcement learning', *arXiv preprint arXiv:1312.5602*.

Perez-Liebana, D. *et al.* (2019) 'General Video Game AI: Competition, Challenges and Opportunities', *IEEE Transactions on Games*, 11(3), pp. 195–214. doi: 10.1109/tg.2019.2901021.

Schaul, T. *et al.* (2015) 'Universal value function approximators', *32nd International Conference on Machine Learning, ICML 2015*.

Schell, J. (2014) *The Art of Game Design: A Book of Lenses, Second Edition*. Taylor \&Francis. Available at: https://books.google.co.id/books?id=kRMeBQAAQBAJ.

Schulman, J. *et al.* (2017) 'Proximal Policy Optimization', *arXiv*. Available at: http://arxiv.org/abs/1707.06347.

Shaker, N., Togelius, J. and Nelson, M. J. (2016) *Procedural Content Generation, Retrogame Archeology*. doi: 10.1007/978-3-319-30004-7_6.

Silver, D. *et al.* (2017) 'Mastering the game of Go wi *Nature*, 550(7676), pp. 354–359. doi: 10.1038/nature24270.

Summerville, A. *et al.* (2018) 'Procedural content generat (PCGML)', *IEEE Transactions on Games*, 10(3), pp. 257–270. doi: 10.1109/TG.2018.2846639.

Susanto, E. K. and Tjandrasa, H. (2021) 'Ap Procedural Level', *2021 3rd East Indonesia Conference on* in *Computer and Information Technology (EIConCIT)*, pp. 427–432.

Sutton, R. S. *et al.* (1999) 'Policy Methods for Reinforcement Learning with Function Approximation', *In Advances in Neural Information Processing Systems 12*. doi: 10.1.1.37.9714.

Sutton, R. S. and Barto, A. G. (1998) *Introduction to reinforcement learning*. MIT press Cambridge.

Togelius, J. *et al.* (2011) 'What is procedural content borderline', *ACM International Conference Proceeding Series*, (May 2014). doi: 10.1145/2000919.2000922.

Togelius, J. and Shaker, N. (2016) 'Search-based', in. doi: 10.1007/978-3-319-42716-4_2.

Volz, V. *et al.* (2018) 'Evolving Mario levels in th convolutional generative', *GECCO 2018 - Proceedings of the 2018 Genetic and Evolutionary Computation Conference*. doi: 10.1145/3205455.3205517.

Watkins, C. J. C. H. and Dayan, P. (1992) 'Q-learning', *Machine Learning*. doi: 10.1007/BF00992698.

Zhou, Z.-H. (2021) 'Ensemble learning', in *Machine learning*. Springer, pp. 181–210.

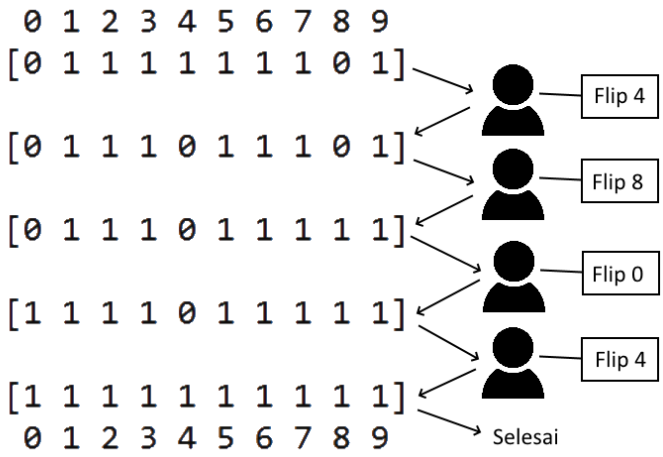
LAMPIRAN 1
UJI COBA DQN + HER MENGGUNAKAN TOY *ENVIRONMENT* DAN *ENVIRONMENT* 1 DIMENSI

Sebelum uji coba dengan menggunakan *environment game* Zelda, arsitektur DQN + HER diuji coba dulu dengan menggunakan 5 buah toy *environment* berbeda. Uji coba ini bertujuan untuk memastikan bahwa arsitektur DQN + HER dapat digunakan untuk melakukan pembuatan *level*. Kelima *environment* yang digunakan dalam penelitian ini dibagi menjadi 2 jenis, yaitu *game* yang *level*-nya direpresentasikan dengan sebuah *array* 1 dimensi dan *game* yang *level*-nya direpresentasikan dengan sebuah *array* 2 dimensi. Masing-masing *game* memiliki aturan permainan dan kriteria yang berbeda. Pada lampiran ini akan dijelaskan tentang aturan permainan, representasi *level*, serta kriteria-kriteria *level* dari *game* yang hendak dibuat. Penjelasan lebih detail dapat dilihat pada (Susanto *and* Tjandrasa, 2021)

Berikut adalah penjelasan tentang 3 *game* yang levelnya direpresentasikan dengan *array* 1 dimensi:

a. *Bit Flipping*

Bit flipping adalah sebuah *game* sederhana yang digunakan untuk menguji performa algoritma *reinforcement learning*. Pada *game* ini, pemain atau AI akan diberi sebuah *array* yang berisi 10 bit. Pemain dapat mengubah sebuah bit dari 0 menjadi 1 atau dari 1 menjadi 0 dengan cara memilih mana bit yang hendak diubah. Pemain harus mengubah semua bit menjadi 1 untuk dapat memenangkan *game* ini.



Ilustrasi permainan dan cara bermain untuk Bit Flipping

Pada penelitian ini akan dibuat sebuah *level generator* yang dapat membuat posisi awal *level* permainan. Pembuat *level* dapat menentukan ada berapa bit yang bernilai 1 pada awal permainan. Representasi *level* pada *game* ini adalah sebuah *array* 1 dimensi yang berisi 10 bit 0 atau 1 yang melambangkan kondisi awal *level*. *Current goal* dari *game* ini adalah sebuah angka yang menyatakan berapa banyak bit pada *level* tersebut yang bernilai 1 sedangkan *desired goal* dari *game* ini adalah sebuah angka yang menyatakan berapa banyak bit bernilai 1 yang harus ada di *level* tersebut. Ilustrasi representasi *level*, *current goal*, dan *desired goal* untuk *level game* ini dapat dilihat pada di bawah. Pada di bawah, nilai *current goal* dapat ditentukan dari representasi *level* sedangkan *desired goal* ditentukan oleh *user* yang meminta sebuah *level* atau dapat diisi *random*.

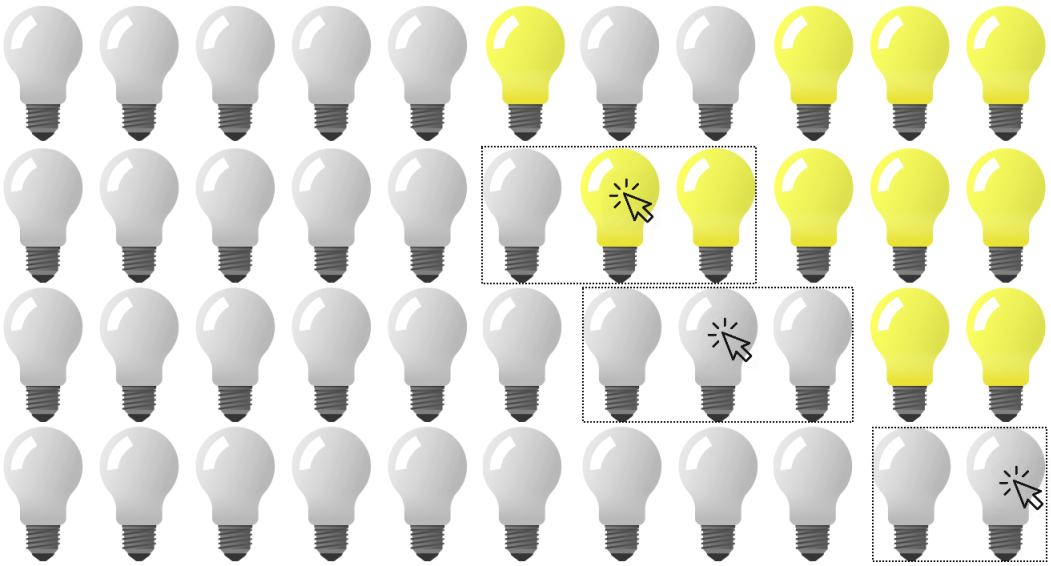
```
Level          : [0 1 0 0 1 1 1 1 0 1]
Current goal   : 6 -> [0 0 0 0 0 0 1 0 0 0 0]
Desired goal   : 5 -> [0 0 0 0 0 1 0 0 0 0 0]

Level          : [1 0 1 1 0 1 1 1 1 0]
Current goal   : 7 -> [0 0 0 0 0 0 0 1 0 0 0]
Desired goal   : 3 -> [0 0 0 1 0 0 0 0 0 0 0]
```

Representasi *level*, *current goal*, dan *desired goal* untuk Bit Flipping (*desired goal* diisi *random*)

Pada *game* ini, *level generator* memodifikasi bit pada *level* satu persatu secara iteratif hingga *desired goal* tercapai. Modifikasi yang dapat dilakukan oleh *generator* adalah mengubah sebuah bit dari 0 menjadi 1 atau dari 1 menjadi 0 dengan cara memilih mana bit yang hendak diubah. Proses pembuatan *level* akan berakhir ketika *current goal* sama dengan *desired goal*.

b. Lights Out

Lights out adalah sebuah *game* di mana seorang pemain diminta untuk mematikan semua lampu yang ada. Pada *game* ini, pemain akan dihadapkan dengan 11 buah lampu dalam kondisi menyala atau mati. Ketika pemain memilih sebuah lampu, maka lampu tersebut akan berubah keadaan dari menyala menjadi mati atau dari mati menjadi menyala. Lampu yang berada di sebelah kiri dan kanan dari lampu yang dipilih juga ikut berubah keadaan dari mati menjadi menyala atau dari menyala menjadi mati. Pemain harus mematikan semua lampu untuk memenangkan *game* ini.



Ilustrasi permainan dan cara bermain untuk Lights Out

Pada penelitian ini akan dibuat sebuah *level generator* yang dapat membuat *level* dengan tingkat kesulitan yang diinginkan. Tingkat kesulitan *level game* ini dapat diukur dari berapa langkah minimum yang diperlukan

untuk menyelesaikan *level* tersebut. Representasi dari *game* ini adalah sebuah *array* 1 dimensi yang berisi 11 bit 0 atau 1. Bit 0 melambangkan lampu yang mati sedangkan bit 1 melambangkan lampu yang menyala. *Current goal* dari *game* ini adalah sebuah angka yang menyatakan langkah minimum yang diperlukan untuk menyelesaikan *level* tersebut sedangkan *desired goal* dari *game* ini adalah sebuah angka yang menyatakan berapa langkah minimum yang diminta. Sama seperti sebelumnya, nilai *current goal* dapat ditentukan dari representasi *level* sedangkan *desired goal* ditentukan oleh *user* yang meminta sebuah *level* atau dapat diisi *random*.

Representasi *level*, *current goal*, dan *desired goal* untuk Lights Out

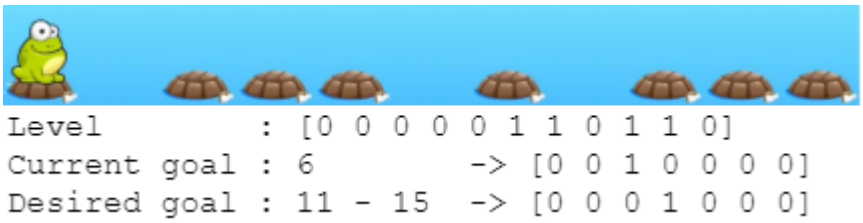
Pada *game* ini, *level generator* memodifikasi keadaan lampu pada *level* satu persatu secara iteratif hingga *desired goal* tercapai. Modifikasi yang dapat dilakukan oleh *generator* adalah mengubah sebuah lampu dari mati menjadi menyala atau dari menyala menjadi mati dengan cara memilih mana lampu yang kondisinya hendak diubah. Proses pembuatan *level* akan berakhir ketika *current goal* sama dengan *desired goal*.

c. *Leaping Frog*

Leaping frog adalah sebuah *game* di mana seorang pemain mengontrol seekor katak yang hendak menyeberangi sebuah sungai. Katak menyeberang dengan cara melompat di atas kura-kura yang ada di sepanjang sungai. Katak yang dikendalikan hanya dapat melompat 1 langkah ke kanan atau 2 langkah ke kanan. Untuk memenangkan *game* ini, pemain harus mengendalikan katak agar dapat menyeberangi sungai tanpa terjatuh ke dalam sungai.

Ilustrasi permainan dan langkah yang dapat diambil oleh pemain dalam permainan Leaping Frog. Pada gambar dapat dilihat bahwa ada 6 cara berbeda bagi katak untuk dapat menyeberangi sungai

Pada penelitian ini akan dibuat sebuah *level generator* yang dapat membuat *level* dengan tingkat kesulitan yang diinginkan. Tingkat kesulitan *level game* ini dapat diukur dari berapa banyak cara yang dapat dilakukan pemain untuk menyelesaikan *level* tersebut. Representasi dari *game* ini adalah sebuah *array* 1 dimensi yang berisi 10 bit 0 atau 1. Bit 1 melambangkan bahwa di posisi tersebut terdapat kura-kura sedangkan bit 0 melambangkan bahwa tidak ada kura-kura di posisi tersebut. *Goal* merepresentasikan banyak cara katak dapat menyeberangi sungai dalam bentuk *array* 1 dimensi berisi 0 atau 1. Angka pertama dari representasi *goal* akan menjadi 1 jika katak tidak bisa menyeberangi sungai. Bit kedua akan menjadi satu jika 1-5 cara berbeda bagi katak untuk menyeberangi sungai. Bit ketiga akan menjadi 1 jika ada 6-10 cara berbeda bagi katak untuk menyeberangi sungai, begitu seterusnya. Bit terakhir akan menjadi 1 jika ada lebih dari 25 cara katak menyeberangi sungai.



Representasi *level*, *current goal*, dan *desired goal* untuk Leaping Frog

Pada *game* ini, *level generator* memodifikasi keadaan lampu pada *level* satu persatu secara iteratif hingga *desired goal* tercapai. Modifikasi yang dapat dilakukan oleh *generator* adalah mengubah sebuah lampu dari mati

menjadi menyala atau dari menyala menjadi mati dengan cara memilih mana lampu yang kondisinya hendak diubah. Proses pembuatan *level* akan berakhir ketika representasi *goal* sama dengan *desired goal*.

Selain 3 *environment* sebelumnya, digunakan pula 2 buah toy *environment* lain. Kedua toy *environment* berikut adalah modifikasi dari *environment* Zelda dengan cara membuang semua jenis *tile* selain *tile* kosong dan *tile* tembok. Kedua *environment* berikut dibuat untuk mengecek apakah algoritma *reinforcement learning* dapat berjalan dengan benar. Berikut adalah penjelasan tentang toy *environment* yang levelnya direpresentasikan dengan *array* 2 dimensi:

a. *Wall*

Pada *environment* wall, agen ditugasi untuk membuat sebuah *level* dengan jumlah tembok sesuai dengan permintaan. *Environment* ini didesain mirip dengan *environment* bit flipping pada bagian sebelumnya. Perbedaannya adalah representasi *level* pada *environment* ini berupa *array* 2 dimensi.

b. *Region*

Pada *environment region*, agen ditugasi untuk membuat mendesain agar sebuah *level* hanya memiliki 1 buah *region*. Definisi *region* telah dijelaskan pada Bab 3. Ketika *environment* menjalankan perintah *reset*(), *environment* akan membuat sebuah *level* berisi petak kosong dan petak tembok dengan jumlah dan posisi *random*. Karena jumlah dan posisi tembok yang *random*, pada *environment* akan terbentuk beberapa buah *region*. Agen kemudian ditugasi untuk membuat agar *level* hanya tinggal memiliki 1 *region* saja.

B. Skenario Uji Coba

Pengujian akan dilakukan dengan cara memberi beberapa kombinasi kriteria yang berbeda dan mengukur berapa persen dari kriteria yang dapat dipenuhi oleh agen. Performansi dari agen hasil *training* akan dibandingkan dengan agen yang bekerja secara *random* untuk menunjukkan bahwa agen yang dilatih mampu membuat *level* dengan kriteria yang diberikan. Performansi agen akan diukur

berdasarkan berapa persen *level* yang dihasilkan yang memenuhi kriteria yang diberikan.

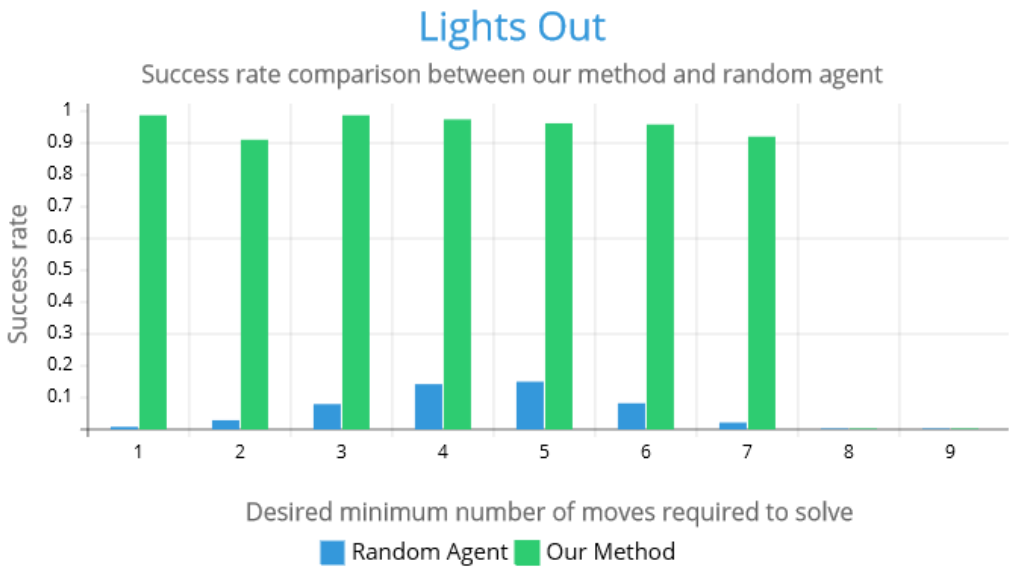
Setiap *game* memiliki kriteria yang berbeda. Berikut adalah penjelasan dari kriteria apa saja yang digunakan dalam uji coba:

1. Bit flipping: Pada *game* ini, pengguna dapat menentukan ada berapa bit yang bernilai 1 pada awal permainan. Banyaknya bit yang boleh bernilai 1 di awal permainan adalah 0, 1, 2, 3, 4, 5, 6, 7, 8, atau 9. *Level generator* akan membuat 1000 *level* untuk masing-masing kriteria kemudian akan dihitung berapa persen *level generator* dapat memenuhi kriteria yang diberikan oleh pengguna.
2. Lights out: Pada *game* ini, pengguna dapat menentukan berapa langkah minimum yang diperlukan untuk menyelesaikan *level* tersebut. Kemungkinan langkah minimum yang diperlukan untuk menyelesaikan *level* adalah 1, 2, 3, 4, 5, 6, 7, 8, atau 9. *Level generator* akan membuat 1000 *level* untuk masing-masing kriteria kemudian akan dihitung berapa persen *level generator* dapat memenuhi kriteria yang diberikan oleh pengguna.
3. Leaping frog: Pada *game* ini, pengguna dapat menentukan range berapa banyak cara menyelesaikan *level* tersebut. Range banyaknya penyelesaian yang disediakan oleh *level generator* ini adalah 1 - 5, 6 - 10, 11 - 15, 16 - 20, 21 - 25, dan >25. *Level generator* akan membuat 1000 *level* untuk masing-masing kriteria kemudian akan dihitung berapa persen *level generator* dapat memenuhi kriteria yang diberikan oleh pengguna.

Untuk kedua toy *environment* yang direpresentasikan dengan *array* 2 dimensi, penelitian tidak dilanjutkan sampai ke tahap uji coba. Hal ini karena kedua *environment* ini hanya digunakan untuk mengetes apakah proses *training* dapat berjalan dengan benar. Selain itu, seperti yang dapat pada Bab 4, *success rate* hasil *training* untuk environmentt ini pada saat *training* naik kemudian turun dan tidak dapat naik kembali.

C. Hasil Uji Coba

Hasil *success rate* untuk *game* bit flipping, lights out, dan leaping frog dapat dilihat pada gambar di bawah. Dapat dilihat bahwa untuk setiap *game*, agen *reinforcement learning* dapat mencapai *success rate* melebihi 90%. Hal ini jauh lebih baik apabila dibandingkan dengan agen *random*. Hal ini menunjukkan bahwa arsitektur DQN + HER yang dirancang mampu mempelajari cara membuat *level* yang memenuhi kriteria yang diminta oleh pengguna. Ada 2 skenario di mana *success rate* dari agen adalah 0 persen. Hal ini karena setelah diselidiki lebih lanjut, ternyata syarat tersebut memang tidak bisa dipenuhi.

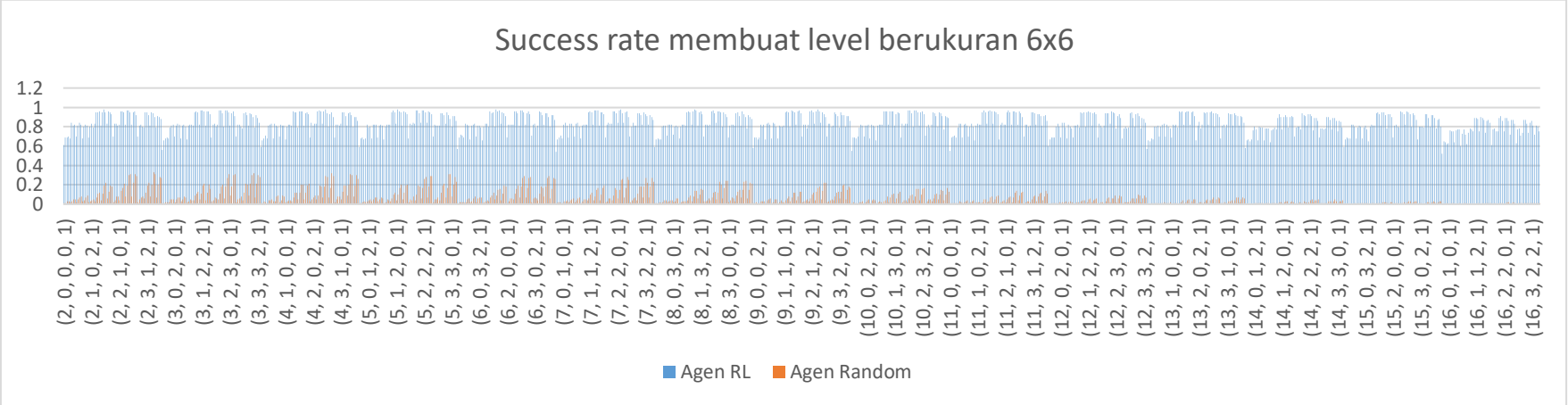

Lights Out
Success rate comparison between our method and random agent
Success rate
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
1
2
3
4
5
6
7
8
9
Desired minimum number of moves required to solve
Random Agent
Our Method


Leaping Frog
Perbandingan success rate metode yang diteliti dengan metode random
Success rate
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
1 - 5
6 - 10
11 - 15
16 - 20
21 - 25
> 25
Banyaknya solusi yang diinginkan
Random
Metode yang diteliti

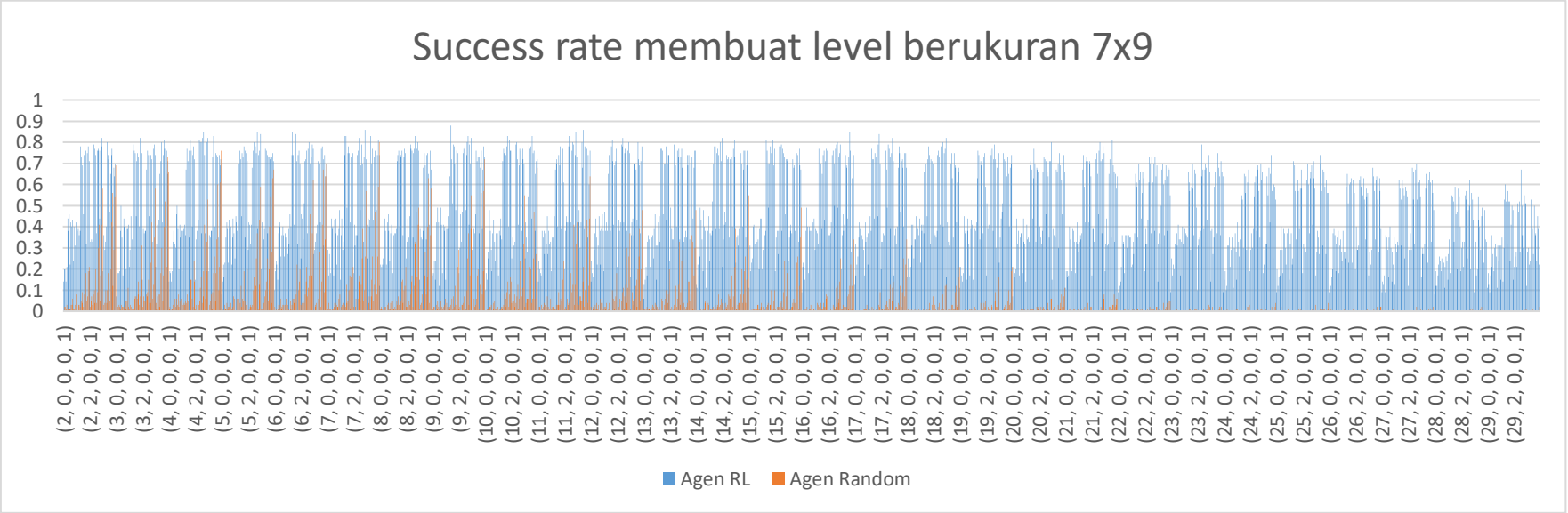
LAMPIRAN 2

SUCCCESS *RATE* AGEN RL UNTUK TIAP KRITERIA YANG DAPAT DIMINTA OLEH PENGGUNA

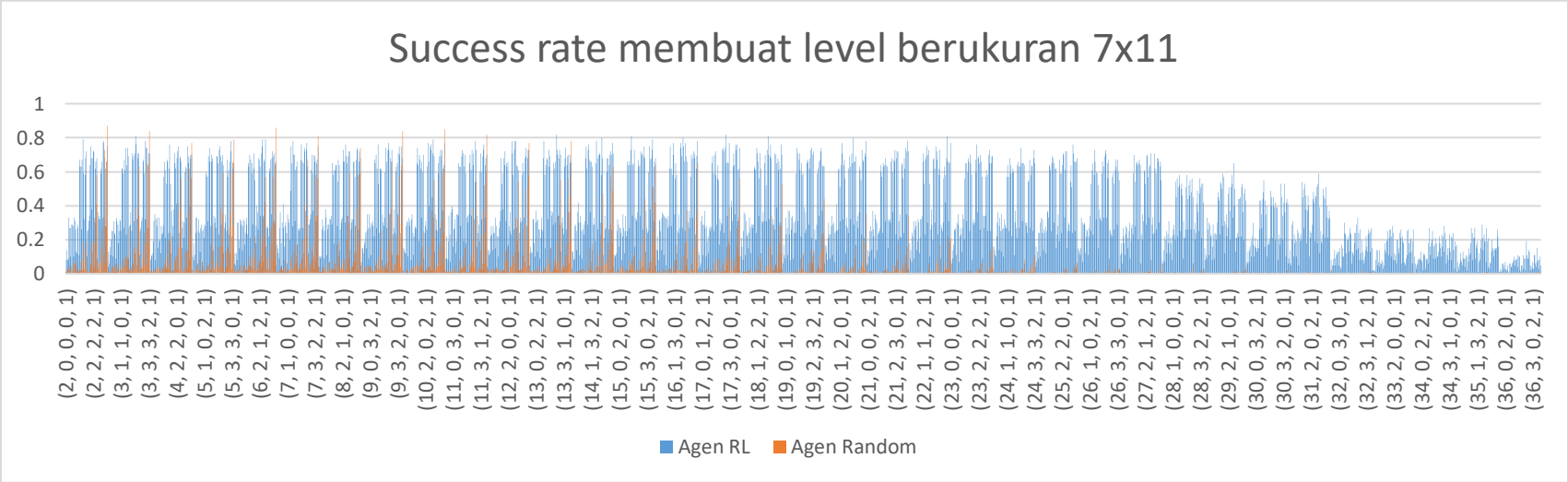
Ukuran *level* 6 × 6



Ukuran *level* 7 × 9



Ukuran *level* 7 × 11



LAMPIRAN 3
PROSES PEMBUATAN *LEVEL*

Pada lampiran ini akan ditunjukkan proses iterasi pembuatan *level-level* yang ditunjukkan pada tabel 4.6, tabel 4.9, dan tabel 4.12. Agar lebih rapi, mudah dibaca, dan menghemat tempat, langkah-langkah iterasi akan diletakkan dalam tabel. Pada masing-masing cell pada tabel akan ditunjukkan keadaan *level* saat itu, *current goal*, lalu aksi apa yang diambil oleh agen *neural network*. Contoh:

Representasi *level*: 

Current goal: **(0, 0, 3, 3, 2)**

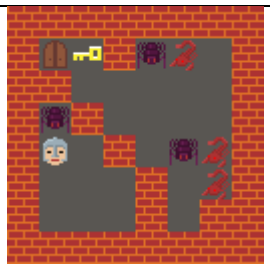
Aksi yang diambil agen: **(0, 0, 4)**

Iterasi ke: **1**

Sebagai pengingat, 5 angka di dalam *goal* merepresentasikan jarak minimum yang diperlukan untuk menyelesaikan *level*, jumlah kelelawar, kalajengking, laba-laba dalam *level*, dan jumlah *region* dalam *level*. Aksi yang diambil oleh agen dinyatakan dalam (y, x, t), dimana y adalah baris, x adalah kolom, dan t adalah nomor *tile*. Nilai dari x dan y dimulai dari 0. Nomor masing-masing *tile* adalah: kosong 0, tembok 1, karakter 2, kunci 3, pintu 4, kelelawar 5, kalajengking 6, dan laba-laba 7.

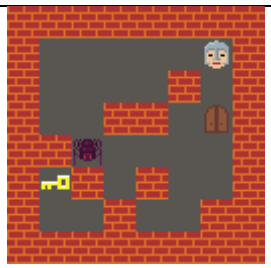
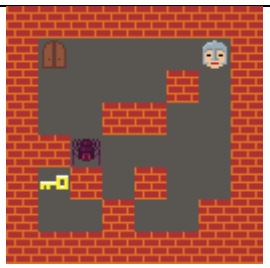
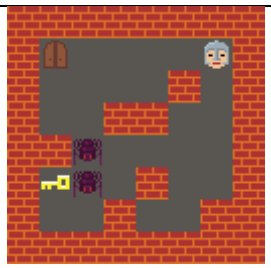
Tabel 4.6 nomor 1

Desired goal: (16, 1, 2, 0, 1)

 (0, 0, 3, 3, 2) (0, 0, 4) 1	 (0, 0, 3, 3, 2) (4, 1, 0) 2	 (0, 0, 3, 3, 2) (0, 2, 5) 3
 (0, 1, 3, 3, 2) (4, 5, 0) 4	 (0, 1, 2, 3, 2) (0, 3, 0) 5	 (0, 1, 2, 2, 2) (1, 0, 0) 6
 (5, 1, 2, 2, 1) (4, 5, 3) 7	 (19, 1, 2, 2, 1) (2, 0, 2) 8	 (18, 1, 2, 1, 1) (3, 4, 4) 9
 (11, 1, 2, 0, 1) (0, 5, 3) 10	 (11, 1, 2, 0, 1) (5, 1, 4) 11	 (18, 1, 2, 0, 1) Selesai

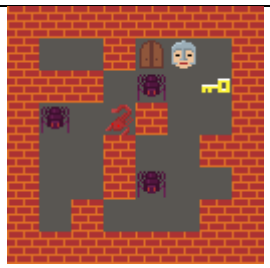
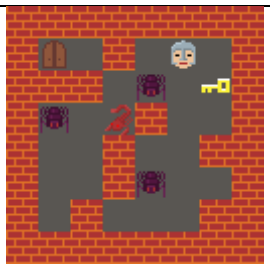
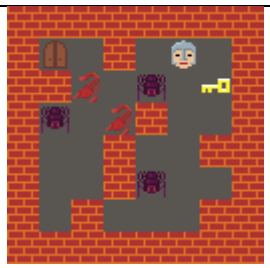
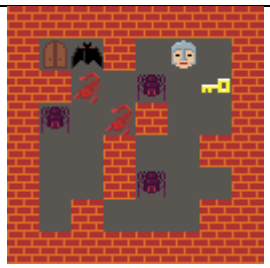
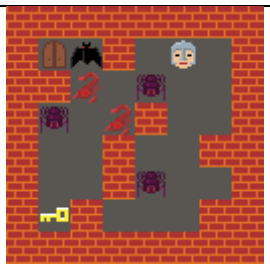
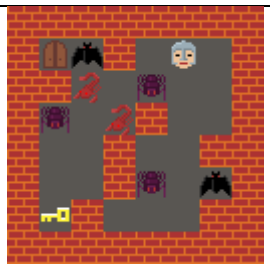
Tabel 4.6 nomor 2

Desired goal: (16, 3, 1, 3, 1)

 (0, 0, 0, 1, 2) (0, 0, 4) 1	 (0, 0, 0, 1, 2) (4, 1, 7) 2	 (15, 0, 0, 2, 1) (0, 1, 5) 3
 (15, 1, 0, 2, 1) (1, 4, 7) 4	 (15, 1, 0, 3, 1) (4, 5, 6) 5	 (15, 1, 1, 3, 1) (0, 3, 5) 6
 (15, 2, 1, 3, 1) (3, 1, 5) 7	 (15, 3, 1, 2, 1) (2, 3, 7) 8	 (15, 3, 1, 3, 1) (5, 0, 3) 9
 (17, 3, 1, 3, 1) Selesai		

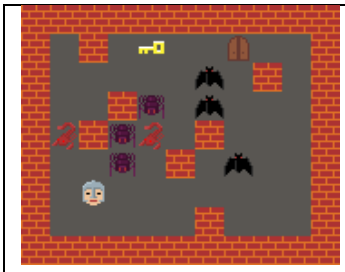
Tabel 4.6 nomor 3

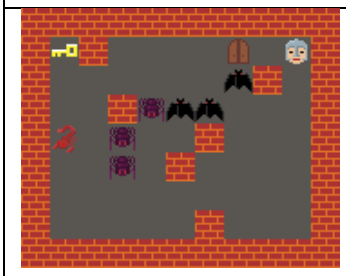
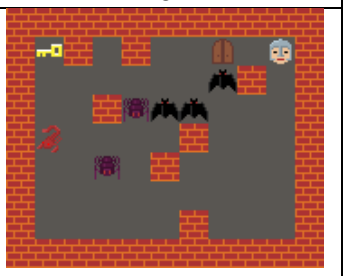
Desired goal: (15, 2, 2, 3, 1)

 (0, 0, 1, 3, 2) (0, 0, 4) 1	 (0, 0, 1, 3, 2) (1, 1, 6) 2	 (8, 0, 2, 3, 1) (0, 1, 5) 3
 (8, 1, 2, 3, 1) (5, 0, 3) 4	 (16, 1, 2, 3, 1) (4, 5, 5) 5	 (16, 2, 2, 3, 1) Selesai

Tabel 4.9 nomor 1

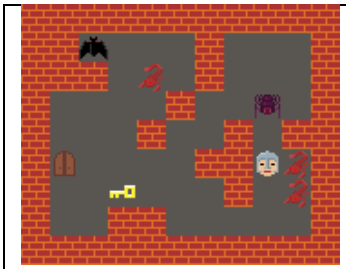
Desired goal: (17, 2, 2, 1, 1)

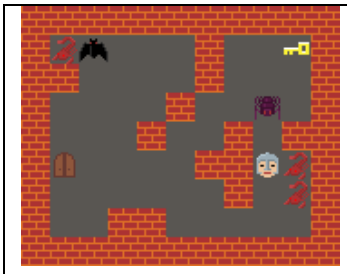
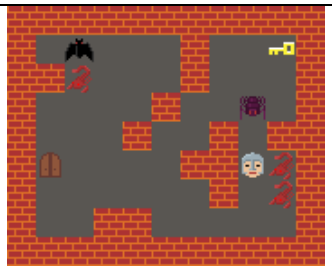
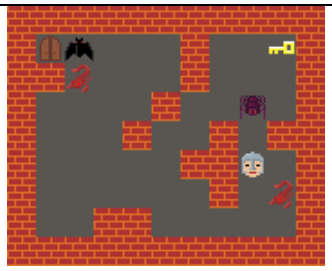
 (10, 3, 2, 3, 1) (3, 1, 0) 1	 (10, 3, 2, 3, 1) (1, 6, 5) 2	 (10, 3, 2, 3, 1) (3, 3, 0) 3
 (10, 3, 1, 3, 1) (0, 0, 3)	 (14, 3, 1, 3, 1) (0, 8, 2)	 (20, 3, 1, 3, 1) (2, 4, 5)

4	5	6
 (20, 3, 1, 3, 1) (3, 2, 0) 7	 (20, 3, 1, 2, 1) (0, 3, 1) 8	 (20, 3, 1, 2, 1) (2, 4, 1) 9
 (20, 2, 1, 2, 1) (2, 8, 6) 10	 (20, 2, 2, 2, 1) (2, 3, 4) 11	 (15, 2, 2, 1, 1) (1, 3, 1) 12
 (25, 2, 2, 1, 1) Selesai		

Tabel 4.9 nomor 2

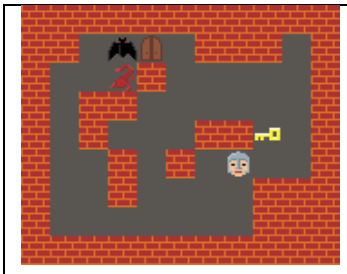
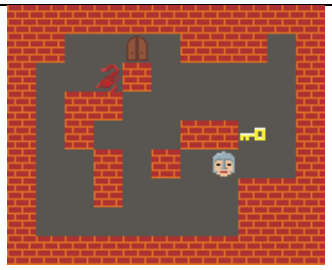
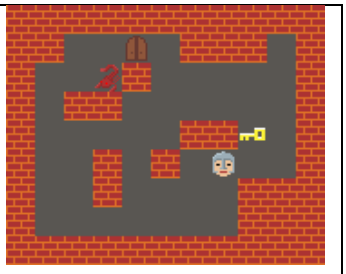
Desired goal: (23, 1, 2, 3, 1)

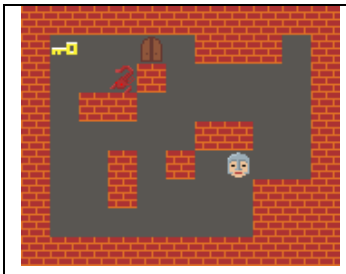
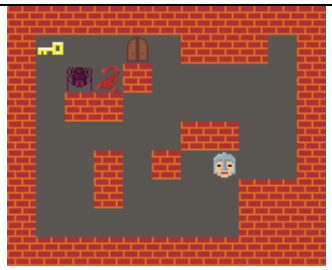
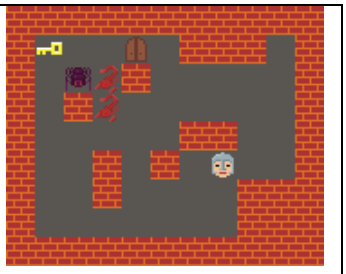
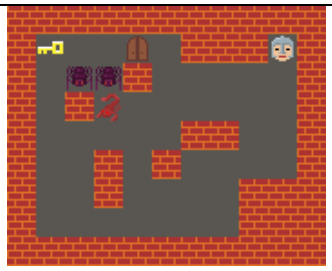
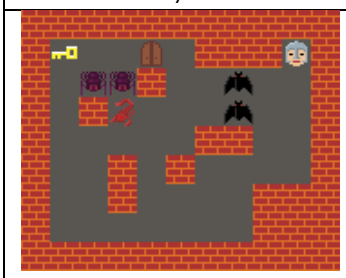
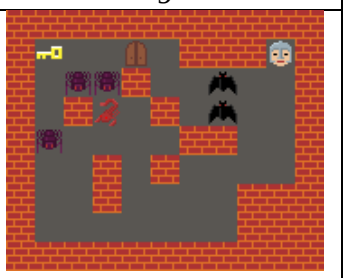
 (11, 1, 3, 1, 1) (0, 8, 3) 1	 (17, 1, 3, 1, 1) (1, 1, 6) 2	 (17, 1, 3, 1, 1) (0, 0, 6) 3

 (17, 1, 3, 1, 1) (1, 1, 6) 4	 (17, 1, 3, 1, 1) (0, 0, 6) 5	 (17, 1, 3, 1, 1) (1, 1, 6) 6
 (17, 1, 3, 1, 1) (0, 0, 4) 7	 (21, 1, 2, 1, 1) (1, 0, 2) 8	 (31, 1, 2, 1, 1) (1, 4, 7) 9
 (31, 1, 2, 2, 1) (2, 4, 7) 10	 (23, 1, 2, 3, 1) Selesai	

Tabel 4.9 nomor 3

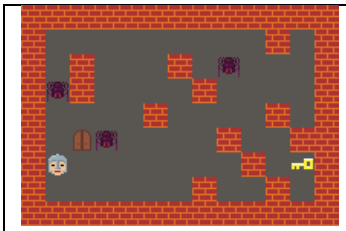
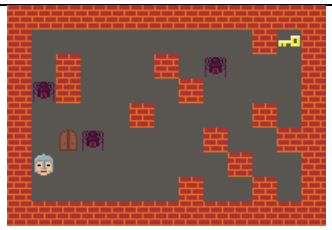
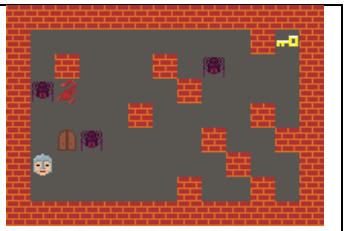
Desired goal: (18, 2, 1, 3, 1)

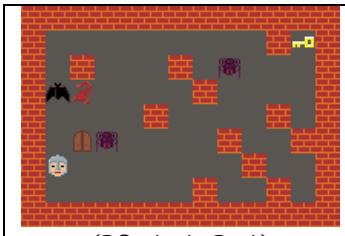
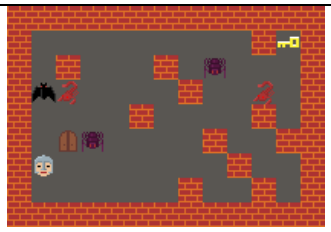
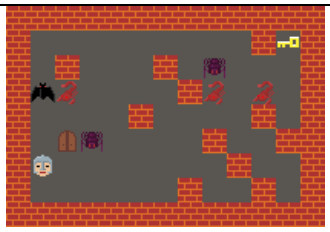
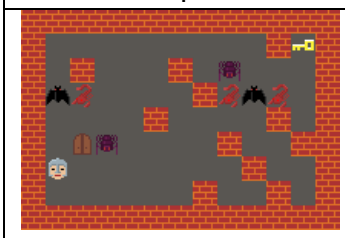
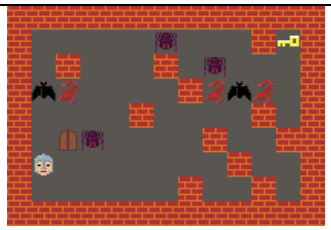
 (9, 1, 1, 0, 1) (0, 2, 0) 1	 (9, 0, 1, 0, 1) (3, 1, 0) 2	 (9, 0, 1, 0, 1) (0, 0, 3) 3

 (15, 0, 1, 0, 1) (1, 1, 7) 4	 (15, 0, 1, 1, 1) (2, 2, 6) 5	 (15, 0, 2, 1, 1) (1, 2, 7) 6
 (15, 0, 1, 2, 1) (0, 8, 2) 7	 (15, 0, 1, 2, 1) (1, 6, 5) 8	 (15, 1, 1, 2, 1) (2, 6, 5) 9
 (15, 2, 1, 2, 1) (3, 0, 7) 10	 (15, 2, 1, 3, 1) (2, 4, 1) 11	 (21, 2, 1, 3, 1) Selesai

Tabel 4.12 nomor 1

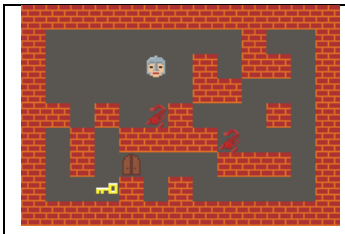
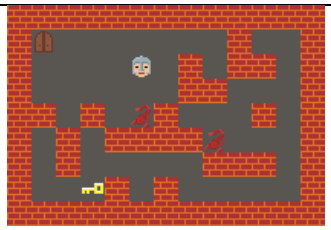
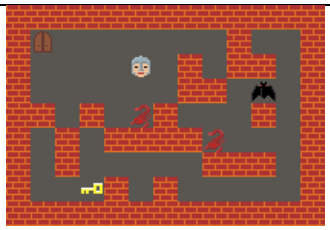
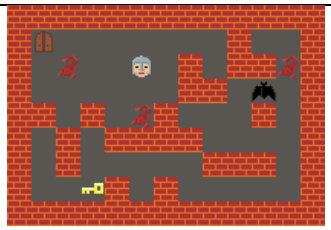
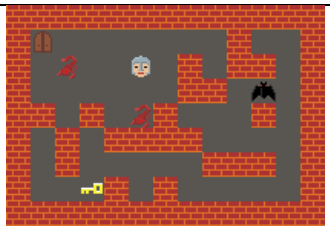
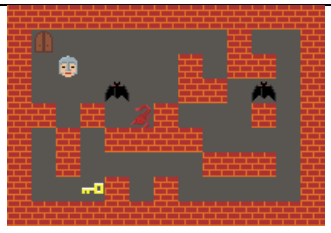
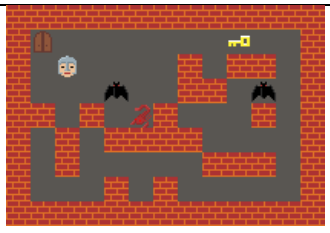
Desired goal: (22, 2, 3, 3, 1)

 (26, 0, 0, 3, 1) (0, 10, 3) 1	 (28, 0, 0, 3, 1) (2, 1, 6) 2	 (28, 0, 1, 3, 1) (2, 0, 5) 3

 (28, 1, 1, 2, 1) (2, 9, 6) 4	 (28, 1, 2, 2, 1) (2, 7, 6) 5	 (28, 1, 3, 2, 1) (2, 8, 5) 6
 (28, 2, 3, 2, 1) (0, 5, 7) 7	 (28, 2, 3, 3, 1) Selesai	

Tabel 4.12 nomor 2

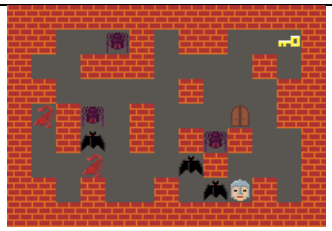
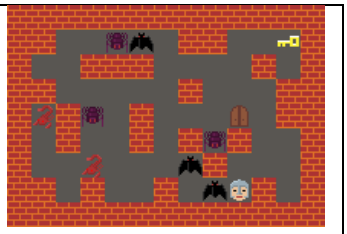
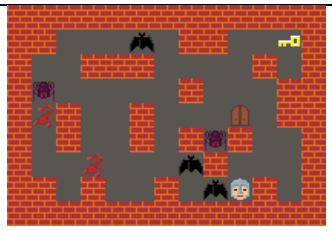
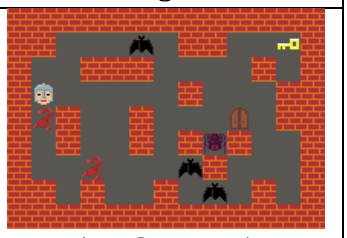
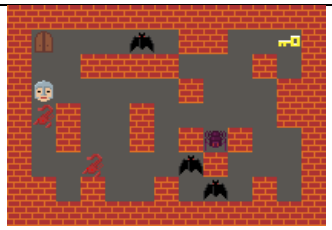
Desired goal: (10, 2, 1, 0, 1)

 (0, 0, 2, 0, 3) (0, 0, 4) 1	 (0, 0, 2, 0, 2) (2, 9, 5) 2	 (0, 1, 2, 0, 2) (1, 10, 6) 3
 (0, 1, 3, 0, 2) (1, 1, 6) 4	 (0, 1, 3, 0, 2) (1, 10, 0) 5	 (0, 1, 2, 0, 2) (1, 1, 2) 6
 (0, 1, 1, 0, 2)	 (0, 2, 1, 0, 2)	 (16, 2, 1, 0, 1)

(2, 3, 5) 7	(0, 8, 3) 8	Selesai

Tabel 4.12 nomor 3

Desired goal: (28, 3, 2, 1, 1)

 (17, 3, 2, 3, 1) (0, 10, 3) 1	 (15, 3, 2, 3, 1) (0, 4, 5) 2	 (15, 3, 2, 3, 1) (2, 0, 7) 3
 (15, 3, 2, 3, 1) (0, 3, 0) 4	 (15, 3, 2, 2, 1) (2, 0, 2) 5	 (17, 3, 2, 1, 1) (1, 5, 1) 6
 (19, 3, 2, 1, 1) (0, 0, 4) 7	 (30, 3, 2, 1, 1) Selesai	

BIOGRAFI PENULIS

Evan Kusuma Susanto, lahir di Surabaya pada tanggal 03 Februari 1996. Penulis telah menempuh pendidikan di SDK Santa Clara Surabaya (2002-2008), SMPK Santa Clara Surabaya (2008-2011), SMAK St. Louis 1 Surabaya (2011-2014).

Setelah menyelasikan pendidikan dasar dan menengahnya, penulis melanjutkan kuliah sarjana di Institut Sains dan Teknologi Terpadu Surabaya jurusan Teknik Informatika (2014-2018). Penulis melanjutkan studi pasca sarjana di Institut Teknologi Sepuluh Nopember Surabaya jurusan Teknik Informatika (2019-2022). Penulis mengambil bidang minat Komputasi Cerdas dan Visualisasi (KCV). Penulis dapat dihubungi melalui email: evanks52@gmail.com