MIND (Multimedia Artificial Intelligent Networking Database) Journal

MIND Journal | ISSN (p): 2528-0015 | ISSN (e): 2528-0902 | Vol. 9 | No. 2 | Halaman 135 - 152
DOI : https://doi.org/10.26760/mindjournal.v9i2.135-152 Desember 2024

Pengujian Parameter Algoritma Genetika dan *Feed-Forward Neural Networks* pada Permainan Ular Klasik

AHMAD BISRY, CECEP MUHAMAD SIDIK RAMDANI, SITI YULIYANTI

Program Studi Informatika, Universitas Siliwangi
Email: 207006056@student.unsil.ac.id

Received 23 Juni 2024 **|** *Revised* 19 Juli 2024 **|** *Accepted* 20 Agustus 2024

ABSTRAK

Konfigurasi parameter yang tepat sangat penting untuk memaksimalkan kinerja dari sebuah algoritma. Algoritma genetika dan neural networks memerlukan pemilihan parameter yang sesuai dalam penggunaannya. Pada permainan ular, performa diukur dari score dan efisiensi runtime. Penelitian ini menguji parameter untuk menemukan konfigurasi optimal bagi kedua algoritma. Permainan ular digunakan sebagai model eksperimen karena metrik kinerja yang jelas, seperti score yang didapat dan beberapa rintangan yang ada. Sebanyak 60 eksperimen dilakukan untuk membandingkan jumlah generasi dan populasi, mutation chance, dan jumlah neuron pada hidden layer. Hasil penelitian menunjukkan konfigurasi dengan generasi lebih besar dari populasi adalah yang paling optimal, menghasilkan score setara dengan generasi dan populasi yang sama besar, namun dengan runtime lebih efisien. Mutation chance 0.1% merupakan yang terbaik dibandingkan dengan 0.2% sampai 0.5%. Selain itu, hidden layer dengan 16 neuron lebih efisien dibandingkan 24 neuron, baik dari segi score maupun runtime.

***Kata kunci**: Algoritma genetika, neural networks, Permainan ular klasik*

ABSTRACT

Appropriate parameter configuration is crucial to maximizing algorithm performance. Genetic algorithms and neural networks require careful parameter selection. In the game of Snake, performance is measured by score and runtime efficiency. This research tests parameters to find optimal configurations for both algorithms. Snake serves as an experimental model due to clear performance metrics such as score and various obstacles. Sixty experiments compare generation and population sizes, mutation chances, and neuron counts in hidden layers. Findings indicate that configurations with larger generations than populations are optimal, yielding scores similar to equal-sized generations and populations but with more efficient runtime. A 0.1% mutation chance outperforms rates of 0.2% to 0.5%. A hidden layer with 16 neurons proves more efficient than 24 neurons in both score and runtime aspects.

***Keywords**: Genetic algorithm, neural networks, classic snake game*

1. PENDAHULUAN

Permainan ular menjadi salah satu game yang banyak digunakan sebagai objek dari penerapan kecerdasan buatan **(Hau Hor, dkk., 2022)**. Banyak algoritma yang digunakan, seperti algoritma genetika **(Białas, 2019)**, Proximal Policy Optimization **(Zhang & Cai, 2020)**, Deep-Q-Network **(Wei, dkk., 2018)**, dan Best First Search **(Kong & Mayans, 2021)**. Dari banyaknya algoritma yang telah diterapkan, algoritma genetika menjadi salah satu algoritma optimasi yang paling cocok dalam mencari solusi terbaik **(Hau Hor, dkk., 2022)**. Algoritma genetika berasal dari sebuah bidang studi yang dikenal sebagai komputasi evolusioner, digunakan untuk menyalin proses reproduksi dan memilih solusi yang paling cocok. Fungsi ini memungkinkan algoritma genetika untuk menemukan solusi atas masalah yang tidak dapat diambil oleh metode lain karena kurangnya fitur **(Carr, 2014)**.

Neural networks merupakan algoritma yang akan dikombinasikan dengan algoritma genetika. *Neural networks* memiliki sistem yang terdiri dari koneksi antara unit-unit pemrosesan informasi yang disebut *neuron*, yang meniru cara kerja otak manusia dalam membawa dan memproses sinyal. *Neural networks* mencoba meniru struktur dan fungsi *neuron* alami dengan menggunakan input, output, dan fungsi aktivasi. Perceptron adalah bentuk paling sederhana dari *neural networks*, terdiri dari satu *neuron* dengan dua input dan satu output **(Shiruru, 2016)**. Dengan demikian, *neural networks* merupakan tambahan yang berguna untuk berbagai algoritma yang digunakan dalam pemecahan masalah. Berbeda dengan algoritma genetika, *neural networks* bukan tipikal algoritma yang dapat berdiri sendiri untuk menyelesaikan masalah yang cukup kompleks **(Hau Hor, dkk., 2022)**. *Neural networks* biasanya dikombinasikan dengan algoritma genetika menjadi evolutionary *neural networks*, seperti pada game ular **(Białas, 2019)**, Game 2048 **(Boris & Goran, 2017)**, Flappy bird **(Mishra, dkk., 2019)**, dan Slither.io **(Miller, dkk., 2019)**.

Pengujian parameter merupakan tahapan eksperimen yang dilakukan secara berulang dengan sistem trial and error untuk menemukan konfigurasi terbaik dari kedua algoritma **(Ma, 2024)**. Konfigurasi parameter yang tepat dapat meningkatkan performa dari algoritma genetika sebagai evolution operator dan neural networks sebagai vision atau arah gerak dari ular pada permainan ular klasik. Algoritma genetika dapat terjebak dalam solusi lokal jika parameternya tidak di-set dengan benar. Pengujian parameter dapat membantu dalam mengidentifikasi nilai parameter optimal yang dapat mengarahkan pada solusi global **(Uthansakul, dkk, 2020)**. Selain itu, pemilihan parameter yang tepat dapat meningkatkan kecepatan dari konvergensi yang berpengaruh pada runtime dari programnya **(Rahul Ramesh Patil, 2023)**. Maka dari itu, pengujian parameter akan dilakukan pada penelitian ini untuk memaksimalkan performa dari algoritma genetika dan neural networks.

Beberapa penelitian telah dilakukan dengan menerapkan algoritma genetika dan neural network pada permainan ular. Penelitian pertama oleh **(Chi Yuen, dkk., 2021)** melakukan empat eksperimen dan menemukan bahwa hasil terbaik didapat pada *mutation* rate 0.05, 20000 populasi, satu *hidden layer*, dan 16 *neuron*. Namun, penelitian ini tidak menyediakan GUI untuk permainan ular. Penelitian kedua oleh **(Hau Hor, dkk, 2022)** juga berfokus pada pengujian parameter tanpa GUI, menemukan bahwa ular memiliki kinerja terbaik dengan nilai parameter menengah untuk panjang blok, persentase mutasi, dan intensitas mutasi, serta rasio kinerja terbaik/buruk 18:2. Penelitian ketiga oleh **(Białas, 2019)** menggunakan satu *hidden layer* dan 6 *neuron*, dengan hasil terbaik pada populasi 1000, tetapi juga tanpa GUI. Penelitian keempat oleh **(B. HalmosiC. Sik-Lányi, 2019)** menyertakan GUI dan menggunakan *feed-forward neural networks* dengan satu *hidden layer*, namun memiliki batasan generasi maksimal 500 yang membatasi objektivitas hasil.

Dari empat penelitian yang sudah dilakukan sebelumnya, beberapa masih memiliki kekurangan yang sama, yaitu tidak menampilkan GUI permainan ular. Kehadiran GUI tentu sangat penting, karena digunakan sebagai bahan evaluasi, seperti bagaimana ular menghindari bahaya yang ada dan bagaimana ular mendapatkan makanannya. Selain itu, pengujian parameter yang dilakukan dengan membandingkan generasi kecil ke terbesar merupakan pengujian yang kurang optimal, karena generasi paling besar sudah pasti mendapatkan *score* yang lebih baik. Berdasarkan kekurangan yang masih ada pada penelitian sebelumnya, program ini akan dibangun dengan GUI dari permainan ular dan dilakukan dalam 4 pengujian yang terdiri dari 60 kali eksperimen. Pengujian ini dilakukan untuk mendapatkan performa yang maksimal dan efisiensi waktu komputasi yang didasarkan pada pemilihan parameter yang tepat dari algoritma genetika dan *neural networks*. Selain dari menghasilkan konfigurasi parameter yang baik, mengetahui interaksi antara parameter pada kedua algoritma juga merupakan salah satu alasan dilakukannya pengujian, seperti parameter apa yang mempengaruhi *score* yang didapat, dan parameter apa yang berpengaruh besar terhadap waktu komputasi program.

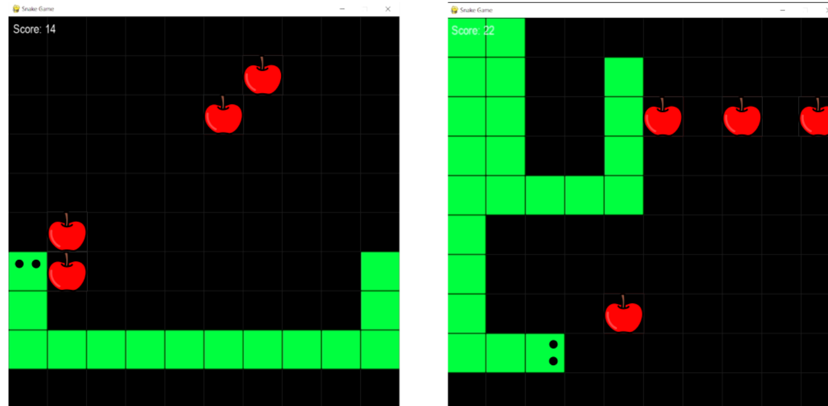
Penelitian yang dilakukan oleh **(Hamdia, dkk, 2021)** menunjukan sebuah peningkatan performa ketika dilakukan pengujian parameter. Hasil pengujian membuktikan bahwa DNN mengungguli ANFIS dengan jumlah generasi GA yang lebih sedikit, menunjukkan efisiensi optimasi. Dari hasil tersebut membuktikan bahwa konfigurasi parameter yang tepat dapat meningkatkan performa dari program. Pada penelitian ini, permainan ular hanya digunakan sebagai model ekperimen AI-nya. Permainan ular memiliki metrik kinerja yang jelas, seperti panjang ular atau skor permainan. Metrik ini memudahkan evaluasi efektivitas konfigurasi parameter yang diuji. Hasil dari penelitian ini berkontribusi dalam memberikan referensi konfigurasi parameter yang tepat, baik untuk algoritma genetika maupun neural networks berdasarkan dari pengujian parameter yang dilakukan. Selain itu, terdapat gambaran bagaimana interaksi dari kombinasi kedua algoritma genetika dalam menemukan dan memecahkan masalah pada permainan ular Klasik.

2. METODOLOGI PENELITIAN

2.1. Perancangan Game

Permainan ular merupakan model eksperimen yang akan digunakan untuk penerapan algoritma genetika dan *feed-forward neural networks*. Perancangan yang dilakukan berkaitan dengan aturan-aturan yang ada pada game yang dapat menjadi metrik uji untuk kedua algoritma. Aturan dari permainan akan disesuaikan dengan Snake AI Competion di Universitas Innopolis **(Brown, dkk, 2021)** dengan sedikit pengembangan, sebagai berikut:

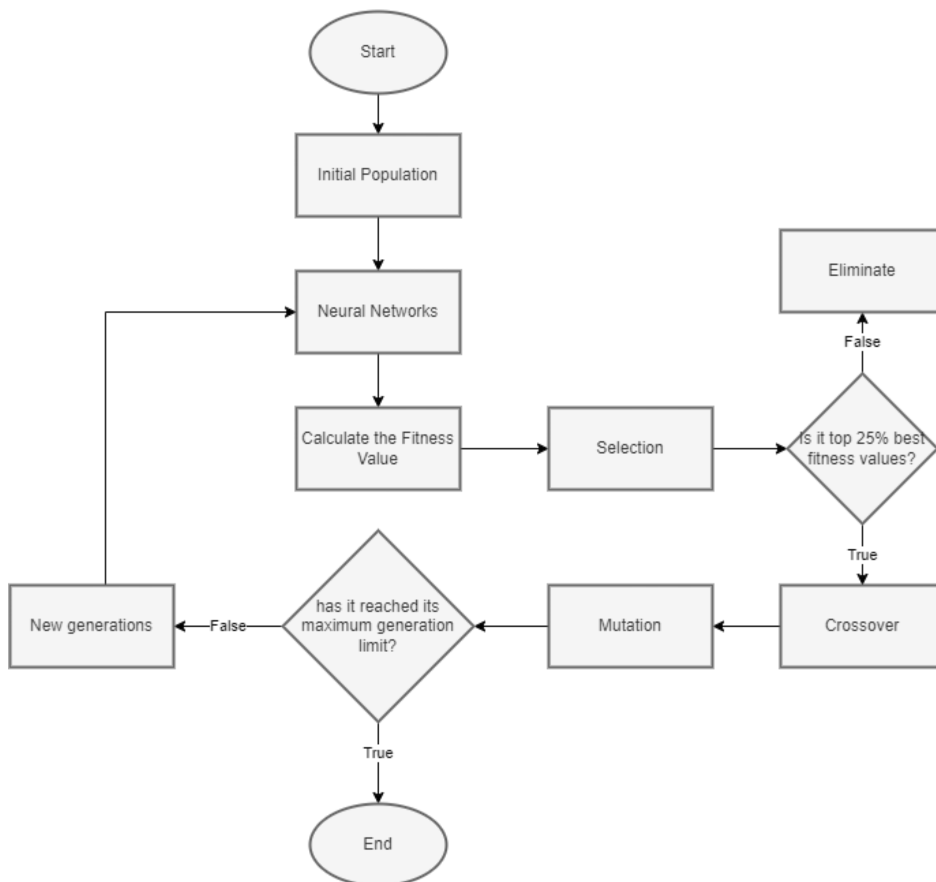
a. Ukuran *board* game 10 x 10 (Gambar 1)
b. Ular hanya dapat bergerak 4 arah, ke atas, ke bawah, ke kanan, dan ke kiri
c. Ular mencari makanan untuk mendapatkan skor dan skor yang didapat membuat panjang ular betambah 1 frame atau piksel
d. Jika ular menabrak badannya sendiri atau mencapai sudut dari *board*, maka game selesai. Ditambahkan fungsi *max_turns* sebagai bahan evaluasi. Ketikaa ular sudah mencapai batas *max_turns*-nya, maka ular akan mati
e. Posisi makanan pada *board* berjumlah 4 dan posisinya spawn secara random



Gambar 1. GUI dari Permainan Ular

2.2. *Developing*

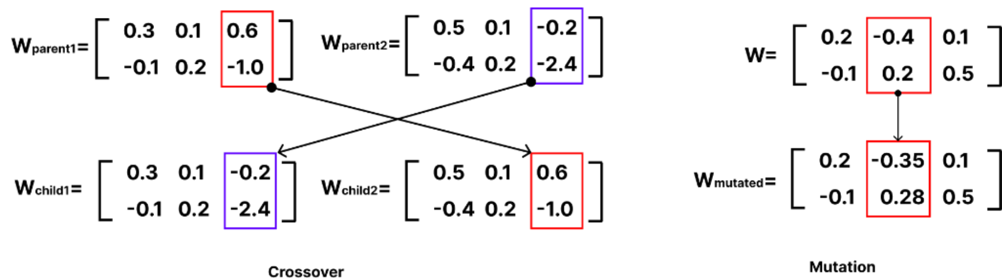
Gambar 2 merupakan flowchart dari implementasi algoritma genetika dan *neural networks* pada permainan ular. Sederhananya, populasi akan dibuat dengan nilai matriks yang acak sebelum masuk ke *neural network*s. Selanjutnya, pada penghitungan nilai fitness masing-masing ular akan bermain sampai kalah sehingga mereka akan mendapatkan skor. Ketika semua ular sudah memiliki skor, dilakukanlah seleksi. Populasi yang terpilih untuk melanjutkan ke tahapan *crossover* dan mutasi hanyalah sebesar 25% terbaik, sedangkan sisanya akan dihapus. Jika pada pengujian jumlah generasi belum sampai batas maksimum yang ditentukan, tahapan tersebut akan dilakukan pengulangan dengan menggunakan generasi yang baru



Gambar 2. Flowchart dari Kombinasi Algoritma Genetika dan *Neural Networks*

2.2.1 Penerapan algoritma genetika

Dalam algoritma genetika, *crossover* dan mutasi adalah dua operator penting yang memainkan peran krusial dalam pengembangan algoritma yang efisien **(Kumar, dkk, 2021)**. Tahapan ini yang memainkan peran penting dalam mencari solusi terbaik pada permainan ular. Ilustrasi bagaimana proses *crossover* dan mutasi ditunjukkan pada Gambar 3.

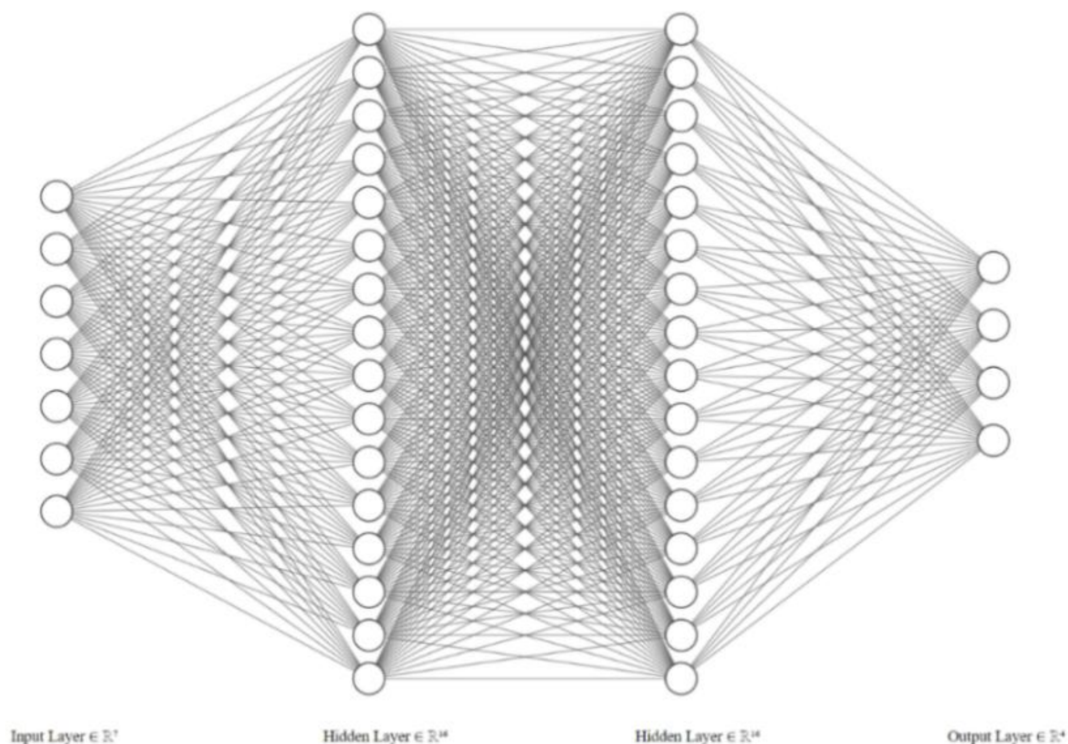


Gambar 3. *Crossover* dan *Mutation* Dapat Merubah Matriks Bobot

Pada operasi *crossover*, kedua *parent* saling bertukar informasi untuk menciptakan bobot *child* yang baru. Bobot *child* yang pertama akan mengambil 2 digit terakhir bobot dari *parent* kedua dan begitu juga sebaliknya. Berbeda dengan *crossover* yang hanya bertukar informasi, pada mutasi ini informasinya berubah sesuai dengan *mutation chance* dan *mutation size* yang di-set. Semakin besar *mutation* size-nya, maka semakin besar juga perubahan informasinya.

2.2.2 Penerapan *feed-forward neural networks*

Jenis *neural networks* yang akan digunakan pada penelitian ini adalah *Feed-forward Neural networks* (FFNN). *Neural networks* yang akan dibangun nantinya memiliki 4 layer, yang terdiri dari 1 input layer, 2 *hidden layer*, dan 1 output layer. Input layer memiliki 7 *neuron*, masing-masing *hidden layer* memiliki 16 *neuron*, dan output layer memiliki 4 *neuron*. Arsitektur dari *neural networks* disajikan pada Gambar 4

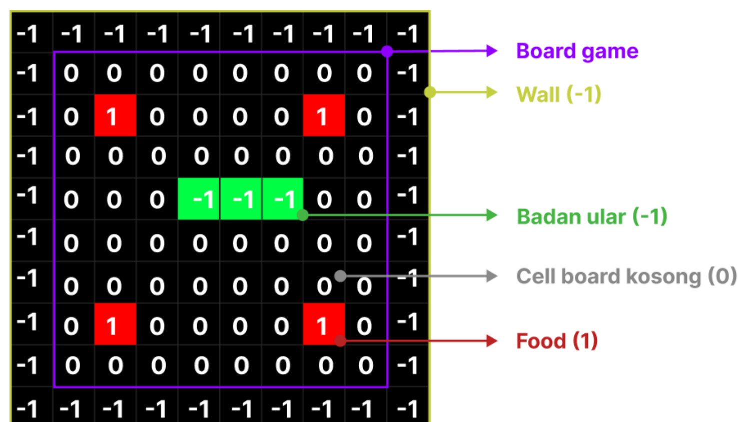


Gambar 4. Arsitektur *Feed-Forward Neural Networks*

Tujuh *neuron* pada *input layer* di-set berdasarkan *window size* dari permainan ular. Jumlah *neuron* pada *hidden layer* dapat berubah sesuai dengan pengujian parameter yang akan dilakukan. *Default* yang di-set adalah 16 *neuron*. Pada *output layer* berisi 4 *neuron* berdasarkan dari arah pergerakan ular yang di mana hanya 4 arah: UP, DOWN, LEFT, RIGHT. Berdasarkan *window size*, banyaknya *neuron* pada input layer adalah 49 (*window_size*^2) + 1 (bias), yang di mana masing-masing *cell* memiliki informasi sebagai berikut:

a. -1 untuk keadaan yang berbahaya seperti dinding dan tubuh ular
b. 1 untuk makanan
c. 0 untuk bagian *cell* yang kosong

Vektor input yang digambarkan pada Gambar 5 akan dimasukan sebagai nilai x pada perhitungan *hidden layer* 1. Hal inilah yang membuat *neural networks* dapat membuat pergerakan ularnya menjadi efisien



Gambar 5. Vektor Input pada *Board* Permainan

2.3. Pengujian Parameter

Setelah algoritma genetika dan *neural networks* berhasil diimplementasikan pada permainan ular, dilakukanlah pengujian parameter dari kedua algoritma. Pengujian ini dilakukan untuk mengetahui interaksi dari masing-masing parameter yang dapat mempengaruhi performa dari ular. Misalnya pada algoritma genetika, bagaimana peran dari generasi dan populasi dalam meningkatkan performa dan juga mempengaruhi efisiensi dari *runtime* programnya. Pengujian parameter terdiri dari 4 pengujian dengan 60 kali eksperimen, sebagai berikut:

a. Pengujian 1 terdiri dari 10 eksperimen dengan jumlah generasi yang besar, yaitu 1000. Namun memiliki populasi yang kecil, yaitu 100.
b. Pengujian 2 terdiri dari 10 eksperimen dengan jumlah generasi yang kecil, yaitu 100. Namun memiliki populasi yang besar, yaitu 1000
c. Pengujian 3 terdiri dari 10 eksperimen dengan jumlah generasi dan populasi sama sama besar, yaitu 1000
d. Pengujian 4 terdiri dari 30 eksperimen: 20 eksperimen menguji masing-masing *mutation chance* 5 kali dari 0.1% sampai 0.5% dan 10 eksperimen menguji jumlah *neuron* pada *hidden layer* (16 dan 24)

2.4 Parameter *Setting*

Gambar 6 menunjukan parameter yang nilainya akan diubah-ubah selama pengujian dilakukan. Nilai yang ditampilkan merupakan nilai default dari programnya. Parameter *max_turns* bersifat absolute yang nilainya akan tetap 200 selama pengujian dilakukan. Tujuan dari adanya *max_turns* ini digunakan untuk bahan evaluasi, apakah ularnya dapat bertahan lama sampai batas turn-nya tanpa menabrak tembok atau memakan badannya sendiri.

```
#Parameter Setting

pop_size = 100
num_generations = 100
hidden_size = 16
mutation_chance = 0.1
max_turns = 200
```

Gambar 6. Default Parameter *Setting* pada Program

3. HASIL DAN PEMBAHASAN

3.1 Perbandingan Generasi dan Populasi

Pada pengujian yang pertama akan dibandingkan mana yang terbaik antara generasi lebih besar dari populasi, populasi lebih besar dari generasi dan generasi sama besar dengan populasi. Perbandingan ini didasarkan pada besarnya *score* dan juga efisiensi dari *runtime* programnya.

3.1.1 Pengujian 1 (generasi lebih besar dari populasi)

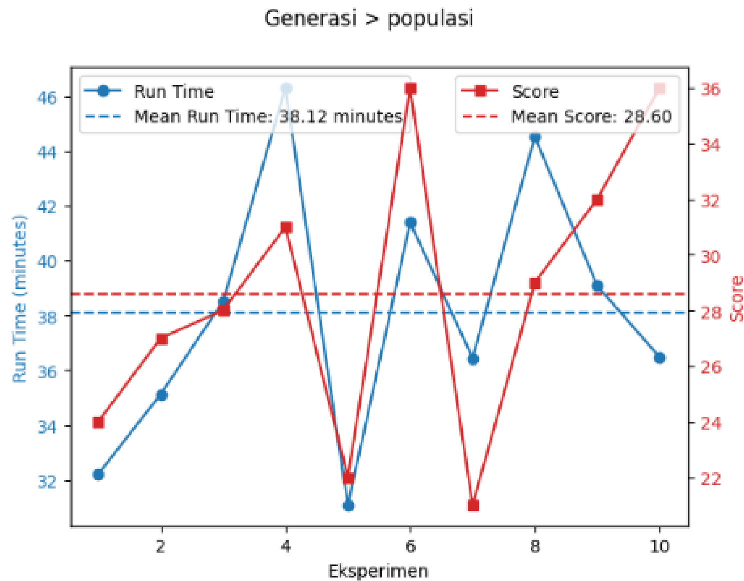
Dengan melebarkan generasi, tentunya algoritma genetika dapat memiliki lebih banyak kesempatan untuk melakukan operasi *crossover* dan *mutation*. Hal ini yang dapat membuat kenaikan performa dari brain ularnya. Selain itu, sedikitnya populasi dapat meminimalkan lamanya *runtime* pada program. Populasi yang lebih kecil berarti setiap generasi lebih cepat dihitung, sehingga iterasi bisa dilakukan lebih cepat. Namun, populasi yang kecil juga bisa membatasi keragaman genetik, meningkatkan risiko konvergensi prematur ke solusi lokal. Gambar 7 menunjukan parameter yang di-set untuk pengujian yang pertama. *Hidden_size*, *mutation_chance* dan *max_turns* merupakan default atau nilai absolut untuk pengujian pertama, kedua, dan ketiga.

```
#Parameter Setting

pop_size = 100
num_generations = 2000
hidden_size = 16
mutation_chance = 0.1
max_turns = 200
```

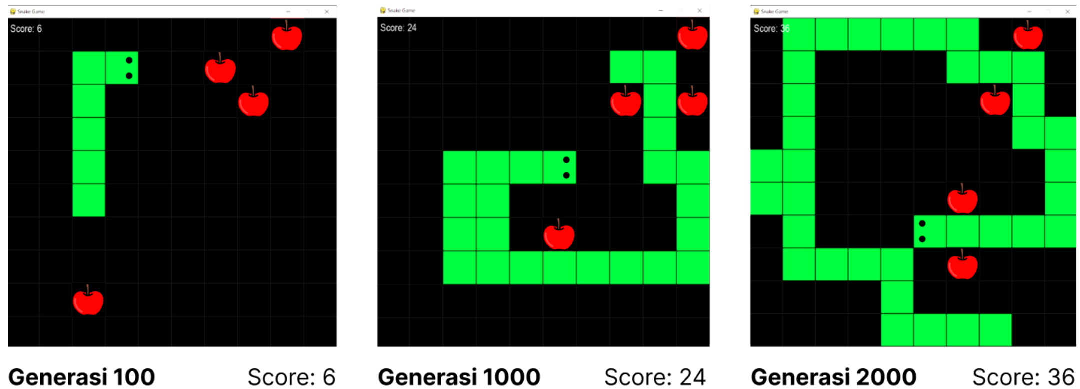
Gambar 7. Parameter *Setting* pada Pengujian 1

Pada pengujian 1 (Gambar 8) yang melebarkan generasi dan meminimalkan populasi, didapatkan rata-rata skor 28,6 dengan rata-rata runtime 38 menit. Skor tertinggi adalah 36 dengan runtime 41,39 menit pada eksperimen ke-6, sedangkan runtime tertinggi adalah 46,30 menit dengan skor 31 pada eksperimen ke-4. Eksperimen ke-4 memiliki runtime tertinggi tetapi tidak menghasilkan skor optimal, kemungkinan karena pengaruh *environment* dalam game. Lamanya runtime dipengaruhi oleh *spawn food* yang acak dan ukuran *board* yang kecil, yang menyebabkan lebih banyak rintangan bagi ular. Namun, ini juga menguntungkan karena *neural networks* bekerja dengan baik, memungkinkan ular bertahan lebih lama meskipun rintangannya banyak.



Gambar 8. Grafik *Score* dan *Runtime* pada Pengujian Pertama

Gambar 9 menunjukan gameplay dari eksperimen ke-6 yang mendapat skor paling tinggi diantara eksperimen yang lainnya. Dapat dilihat ketika ular mendapat skor lebih dari 30, *board*-nya sudah hampir dipenuhi dengan badan ular itu sendiri. Hal inilah yang menjadi salah satu alasan lamanya *runtime*. Ular membutuhkan lebih lama lagi waktu untuk mengeksplorasi dan menghindari rintangan, terutama jika mencapai batas maksimal dari turn-nya.



Gambar 9. Gameplay Pengujian 1 pada Eksperimen ke-6

3.1.2 Pengujian 2 (generasi lebih kecil dari populasi)

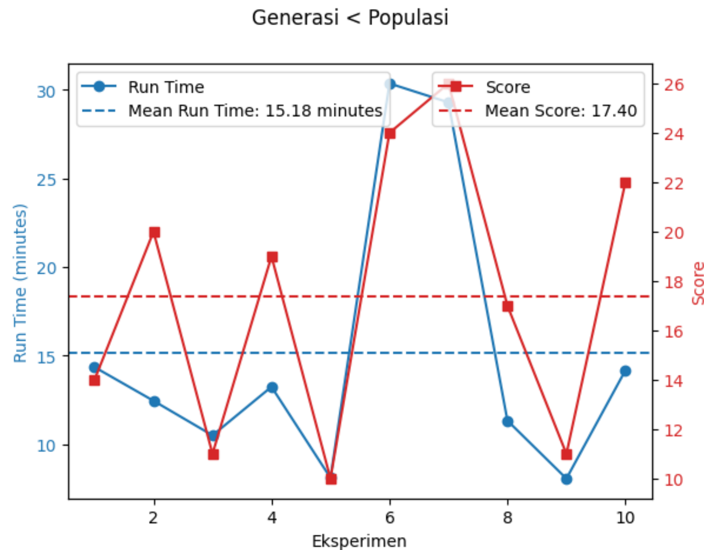
Semakin besarnya populasi dapat meningkatkan keragaman genetik dan kemungkinan menemukan solusi yang lebih baik dalam satu generasi. Selain itu, dapat mengeksplorasi lebih banyak solusi yang memiliki potensial di awal proses. Namun, melebarnya populasi membutuhkan lebih banyak waktu dan sumber daya komputasi per generasinya. Pertimbangan ini yang menjadi alasan dilakukannya penelitian, apakah keragaman genetik yang ada dapat diiringi dengan peningkatan performa atau hanya sekedar memakan lebih banyak waktu untuk komputasinya saja. Semua parameter disesuaikan dengan pengujian yang pertama, kecuali untuk ukuran dari generasi dan populasinya (Gambar 10)

```
#Parameter Setting

pop_size = 2000
num_generations = 100
hidden_size = 16
mutation_chance = 0.1
max_turns = 200
```

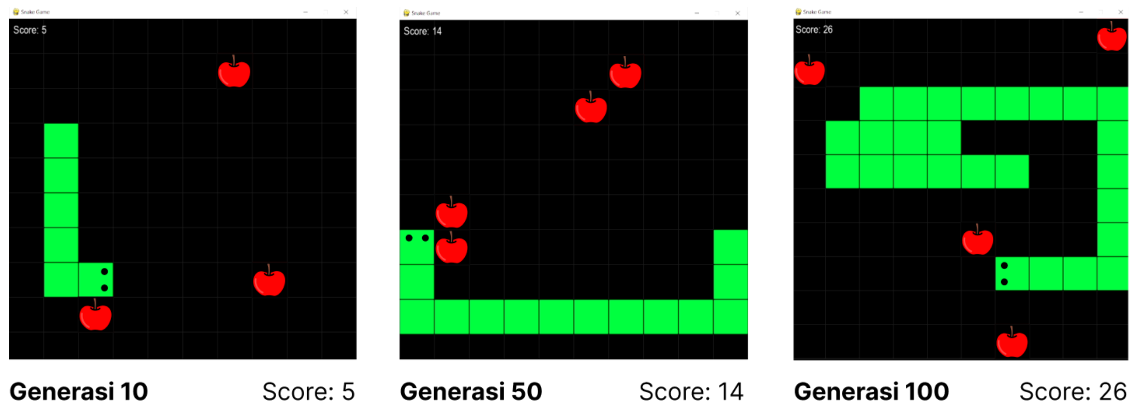
Gambar 10. Parameter *Setting* pada Pengujian 2

Hasil eksperimen dari pengujian 2 dapat dilihat pada Gambar 11. Rata-rata *score* yang didapat adalah 17.40 dan rata rata *runtime*-nya adalah 15.18 menit. Hal ini tentu bukan sebuah hasil yang baik dari segi performa. Selain rendahnya *score* yang didapat, rentang *score* antara yang tertinggi dan yang terendah sangat berbeda jauh, sehingga ini menciptakan inkonsistensi performa. *Score* tertinggi yang didapat ada pada eksperimen ketujuh dengan nilai 26, dan eksperimen kelima menunjukan *score* yang paling rendah, yaitu 10. Rendahnya *score* yang didapat ini diakibatkan dari sedikitnya generasi. Ketika program melakukan reproduce atau regenerate generasi, di sinilah algoritma genetika dapat memaksimalkan fungsinya. Sedikitnya generasi membuat membuat algoritma genetika tidak memiliki waktu yang banyak untuk mengeksploitasi informasi yang didapat dari seleksi dan mutasi. Maka dari itu, ular hanya mengandalkan *neural networks* untuk mendapatkaan *score*nya. Ketidakseimbangan peran dari kedua algoritma ini membuat performa dari ular kurang baik.



Gambar 11. Grafik *Score* dan *Runtime* pada Pengujian 2

Gambar 12 menunjukan salah satu gameplay dari eksperimen yang dilakukan. Pada eksperimen tersebut, *environment* yang menguntungkan membuat ular mendapatkan *score* yang lebih tinggi dibandingkan dengan eksperimen lainnya. Salah satunya, yaitu posisi dari *spawn food* yang berdekatan.



Gambar 12. Gameplay Ular pada Eksperimen ke-7

3.1.3 Pengujian 3 (generasi sama besar dengan populasi)

Pengujian ketiga ini merupakan gabungan dari pengujian pertama dan pengujian kedua. Ukuran dari generasi dan populasi yang sama besar memberikan keuntungan dari segi eksploitas dan eksplorasi. Melebarkan ukuran dari popilasi sehingga sama besar dengan generasi tentu akan menyebabkan lamanya *runtime* dari program seperti yang sudah disebutkan sebelumnya. Namun, apakah lamanya *runtime* ini diiringi dengan performa yang baik dibandingkan dengan dua pengujian yang sebelumnya dan seberapa besar perbedaan *runtime*-nya.

Berbeda dengan pengujian sebelumnya yang menggunakan 2000 baik untuk ukuran populasi maupun generasi. Pada pengujian ketiga digunakan sebanyak 1000 untuk ukuran populasi dan ukuran generasinya (Gambar 13). Bukan tanpa alasan, hal ini didasarkan pada ukuran *board* dan juga *runtime* pada programnya. Dengan jumlah yang sama sebanyak 1000, *score* yang didapat dipastikan sudah bisa mencapai 30. Pelebaran menjadi sama besar 2000 hanya mengakibatkan lamanya *runtime* tanpa menghasilkan objektifitas. Ketika ular mencapai *score* 30, *board*-nya akan semakin penuh dan akan lumayan sulit bagi ular untuk mendapatkan *score* lagi. Inilah alasan mengapa generasi dan populasi yang digunakan hanya sekitar 1000 saja tidak sampai 2000. Selain itu, penggunaan 1000 untuk generasi dan populasi ini tidak terlalu berpengaruh besar terhadap hasil yang akan dibandingkan dengan pengujian 1 dan 2. Pertimbangan ini berdasarkan percobaan sederhana sebelum dilakukannya eksperimen.

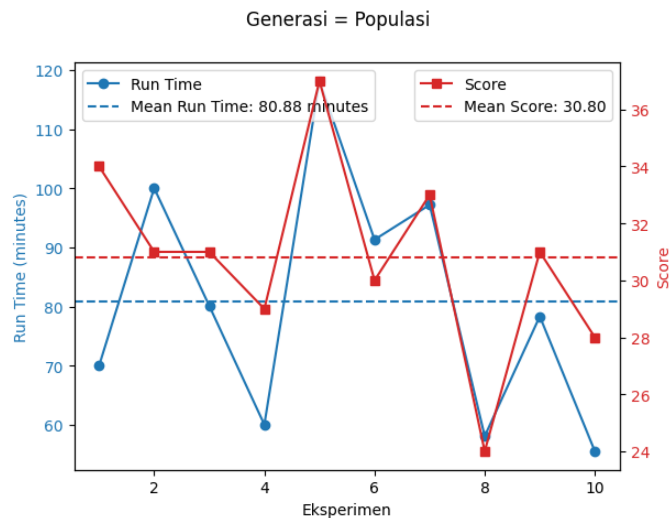
```
#Parameter Setting

pop_size = 1000
num_generations = 1000
hidden_size = 16
mutation_chance = 0.1
max_turns = 200
```

Gambar 13. Parameter *Setting* pada Pengujian 3

Pengujian 3 yang ditunjukan pada Gambar 14 menunjukan rata-rata *score* yang sangat baik dibandingkan dengan kedua pengujian sebelumnya. Namun, rata-rata dari *runtime*-nya juga merupakan yang paling tinggi. Rata-rata *score* yang didapat adalah 30.80, ini merupakan angka yang bisa dibilang luar biasa karena 7 dari 10 eksperimen bisa mendapatkan *score*

diatas 30. Bukan hal yang mudah untuk ular dapat bertahan bahkan menambah *score* ketika sudah mencapai angka 30. Performa yang luar biasa ini dikarenakan hasil eksplorasi dari keragaman genetik berdasarkan besarnya populasi dan optimasi dari operasi *crossover* dan mutasi berdasarkan besarnya generasi.



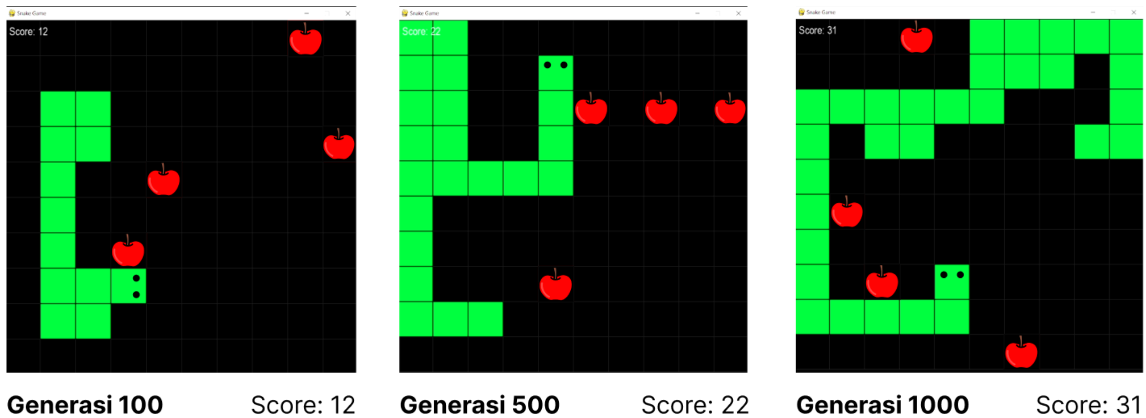
Gambar 14. Grafik *Score* dan *Runtime* pada Pengujian 3

Performa yang baik untuk pengujian 3 ini, membuat *runtime* dari program menjadi sangat lama. Perbedaan *runtime* dari pengujian 3 sangat jauh sekali dibandingkan dengan dua pengujian sebelumnya. Seperti yang sudah dikatakan sebelumnya, ini diakibatkan dari melebarnya populasi yang membutuhkan lebih banyak waktu dan sumber daya komputasi per generasi karena ukuran populasi yang besar. Inilah yang akan terjadi jika pengujian 2 dilakukan pelebaran generasi, apalagi jika dilakukan hampir mendekati ukuran dari populasi.

Berdasarkan penelitian yang dilakukan pada pengujian 3, untuk generasi awal 100-500 waktu komputasinya berjalan dengan normal. Namun, ketika generasi sudah memasuki 600 terjadi perlambatan waktu komputasi. Selain dari besarnya populasi, melambatnya waktu komputasi diakibatkan juga dari semakin besarnya *score* ular. Gambar 15 merupakan gambaran dari ukuran populasi dalam 1 generasi. Angka-angka yang ada pada array tersebut merupakan *score* yang didapat oleh masing-masing ular. Inilah yang membuat waktu komputasi menjadi lambat karena jika ular-ular tersebut sudah mencapai skor yang tinggi, maka dibutuhkan juga waktu yang lebih lama untuk eskplorasi dikarenakan *board* yang sudah penuh dengan badan ular itu sendiri. Jika masing-masing dari 1000 ular tersebut bermain sampai mencapai batas *max_turns*-nya, tentu ini akan semakin memperlambat waktu komputasi karena terlambatnya melakukan regenerasi.

Gambar 15. 1000 Populasi pada Masing-Masing Generasi

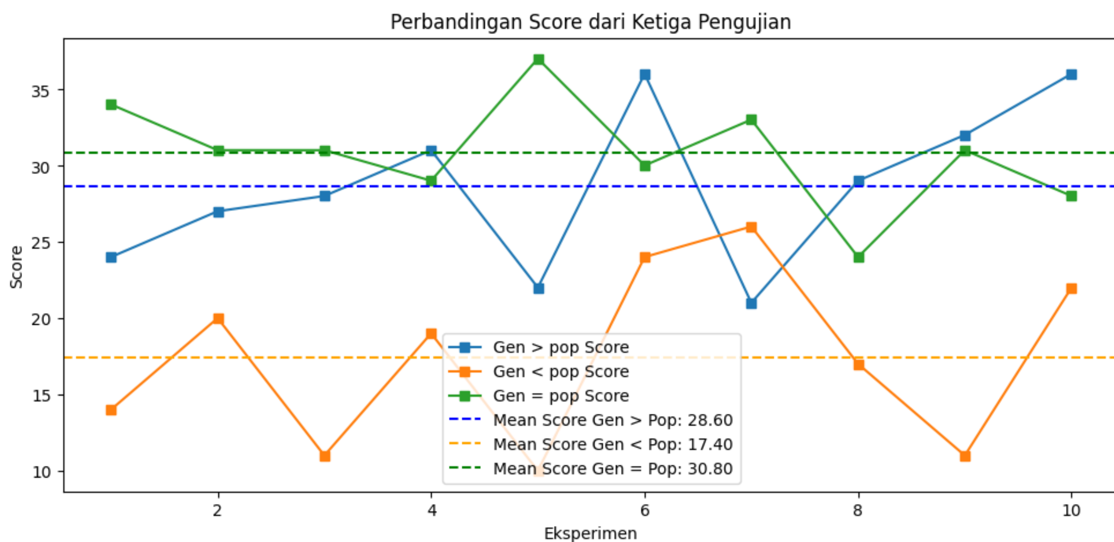
Gambar 16 menunjukan gameplay ular dari pengujian 3 pada eksperimen kedua. Ketika tubuh ular sudah memenuhi *board*, ini mengakibatkan vektor input dari *board* kebanyakan bernilai -1 yang menandakan adanya bahaya. Hal ini mengakibatkan dibutuhkan waktu yang lebih lama untuk komputasinya. Ular akan terus eksplorasi mencari *cell* dari *board* yang tidak berpotensi bahaya. Berbeda dengan yang *score*nya masih dibawah 20 di mana masih terdapat banyak *cell board* kosong yang memberikan vektor input 0, sehingga waktu komputasi normal bahkan bisa saja lebih cepat. Inilah alasan mengapa pada awal generasi komputasinya normal, namun ketika memasuki generasi 600 waktu komputasinya menjadi melambat.



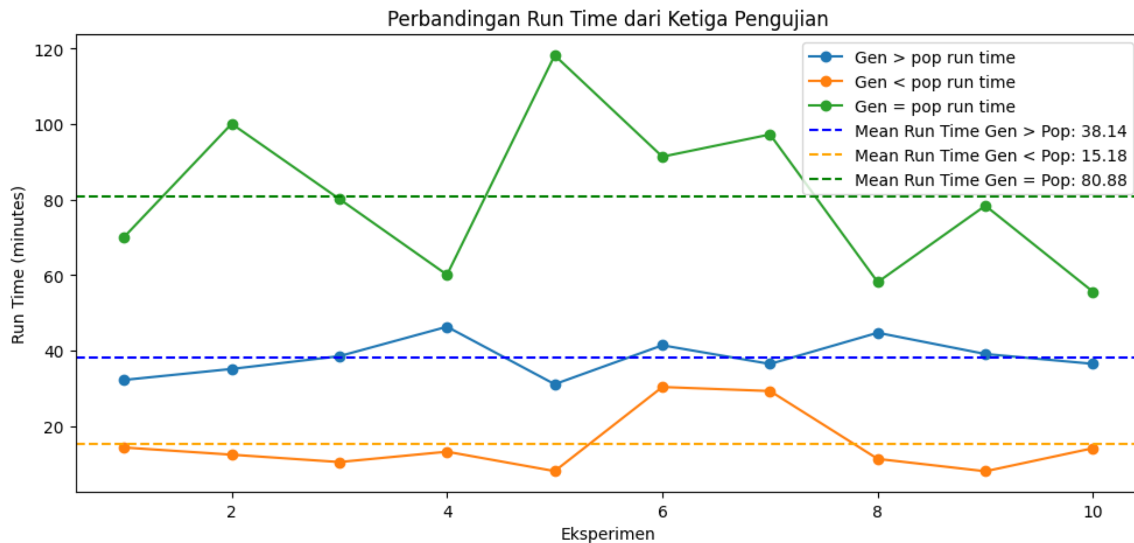
Gambar 16. Gameplay Ular pada Eksperimen ke-2

3.1.4 Evaluasi hasil pengujian

Berdasarkan keseimbangan dari *score* dan *runtime* (Gambar 17 dan Gambar 18), pengujian 1 merupakan parameter *setting* yang paling baik. Rata-rata *score* yang didapat pada pengujian 1 hampir dapat mengimbangi rata-rata *score* dari pengujian ketiga dengan selisih *runtime* yang sangat jauh berbeda. Jika saja generasi pada pengujian pertama dilebarkan sedikit antara 3000-4000, tentu akan mengimbangi rata-rata *score* dari dari pengujian 3 bahkan bisa saja lebih baik tanpa harus mengorbankan *runtime* dengan melebarkan populasi.



Gambar 17. Perbandingan *Score* dari Ketiga Pengujian



Gambar 18. Perbandingan *Runtime* dari Ketiga Pengujian

Pengujian 2 tidak direkomendasikan karena menghasilkan skor yang kecil dan tidak konsisten akibat minimnya generasi yang dibuat. Pelebaran generasi mungkin meningkatkan performa, tetapi akan memperpanjang waktu komputasi dan regenerasi. Meningkatkan populasi hingga 3000 atau 4000 tidak efektif, karena optimasi terbaik terjadi pada regenerasi melalui operasi *crossover* dan mutasi. Pengujian 3 memperoleh rata-rata skor 30.8 dan rata-rata waktu 80.88 menit, yang memuaskan dengan 1000 generasi dan 1000 populasi. Meskipun memperbesar generasi dan populasi menjadi 2000 dapat meningkatkan performa hingga skor 40, waktu komputasi menjadi pertimbangan utama. Dengan 1000 generasi dan populasi, waktu rata-rata sudah mencapai 80.88 menit, dengan waktu tertinggi 118.12 menit pada eksperimen kelima yang menghasilkan skor 37. Meningkatkan jumlah generasi dan populasi akan sangat memakan waktu.

Kesimpulannya, pengujian 1 yang di mana generasi lebih besar dari populasi merupakan parameter *setting* terbaik untuk saat ini berdasarkan pengujian yang dilakukan. Keseimbangan antara *score* yang didapat dan efisiensi *runtime* menjadi faktor utamanya. Pengujian 2 sangat tidak direkomendasikan karena kurangnya generasi membuat algoritma genetika tidak dapat berfungsi secara maksimal, sehingga terdapat inkonsisten performa. Pengujian 3 memiliki kekurangan dari segi *runtime* yang terlalu lama karena melebarnya populasi. Hal ini tentunya akan sangat memakan waktu, terutama jika ularnya sudah tidak menghasilkan objektifitas.

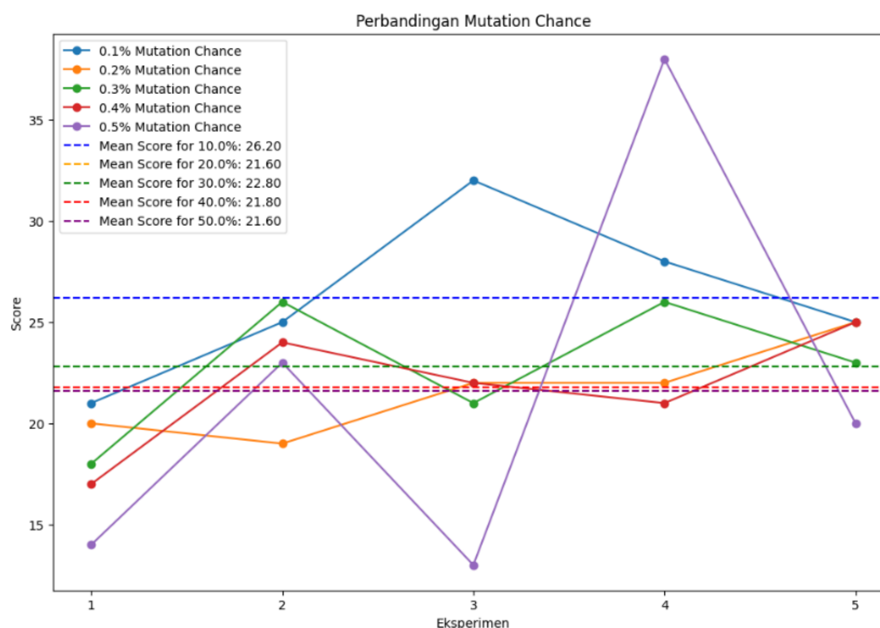
3.2 Perbandingan *Mutation Chance*

Mutation merupakan salah satu fungsi operasi pada algoritma genetika pada tahapan seleksi. Pengujian untuk nilai *mutation chance* dalam algoritma genetika sangat penting karena nilai ini mempengaruhi keseimbangan antara eksplorasi (mencari solusi baru) dan eksploitas (memperbaiki solusi yang ada). *Mutation* bersifat eksplorasi yang di mana perubahan bobot matriks ini diharapkan dapat menemukan informasi yang baru untuk *children* di generasi yang berikutnya. Namun seberapa besar baiknya nilai dari *mutation chance* ini agar tetap seimbang dan tidak menganggu operasi dari *crossover* yang berperan penting dalam membawa informasi dari *parent* yang sebelumnya. Pengujian ini akan dilakukan dalam 30 kali eksperimen yang membandingkan masing-masing nilai *mutation chance*, yaitu 0.1%, 0.2%, 0.3%, 0.4%, dan 0.5%. Berbeda dengan pengujian sebelumnya pada perbandingan generasi dan populasi, untuk pengujian *mutation chance* ini akan mengabaikan *runtime* dan hanya berfokus pada *score* yang didapat. Tidak adanya perbedaan parameter yang mempengaruhi *runtime* pada

pengujian kali ini yang membuat *runtime* diabaikan. Selain itu, sudah dilakukan percobaan sebelumnya bahwa perbedaan dari *mutation chance* tidak berdampak signifikan terhadap *runtime* programnya.

Ukuran populasi yang digunakan adalah 100 dan banyaknya generasi hanya 1000 saja. Hal ini berdasarkan dari pengujian 1 yang dapat memaksimalkan *score* dan memiliki efisiensi *runtime* yang baik. Berbeda dengan pengujian sebelumnya pada perbandingan generasi dan populasi, untuk pengujian *mutation chance* ini akan mengabaikan *runtime* dan hanya berfokus pada *score* yang didapat. Tidak adanya perbedaan parameter yang mempengaruhi *runtime* pada pengujian kali ini yang membuat *runtime* diabaikan. Selain itu, sudah dilakukan percobaan sebelumnya bahwa perbedaan dari *mutation chance* tidak berdampak signifikan terhadap *runtime* programnya.

Eksperimen pada masing-masing *mutation chance* dilakukan hanya sebanyak 5 kali. Dapat dilihat bahwa rata-rata *score* dari *mutation chance* 0.2%, 0.3%, 0.4%, dan 0.5% menghasilkan angka yang tidak begitu jauh. *Mutation chance* 0.1% menghasilkan rata-rata *score* yang paling tinggi di antara yang lainnya. Untuk melihat perbandingan lebih jelasnya dapat dilihat pada grafik Gambar 19.



Gambar 19. Grafik *Score* pada Pengujian *Mutation Chance*

Pengujian pada *mutation chance* 0.1% menghasilkan rata-rata *score* tertinggi dibandingkan dengan yang lainnya. Dengan *mutation chance* yang sangat rendah (0.1%), ular (matriks bobot) dalam populasi cenderung mengalami perubahan yang sangat sedikit dari satu generasi ke generasi berikutnya. Hal ini dapat menjaga stabilitas evolusi, memungkinkan solusi yang baik dipertahankan dan disempurnakan secara bertahap tanpa terlalu banyak gangguan dari mutasi yang besar. Selain itu, *mutation chance* yang rendah memungkinkan keseimbangan yang baik antara eksplorasi ruang solusi baru dan eksploitasi solusi yang sudah ada, di mana dapat menghasilkan performa yang lebih baik.

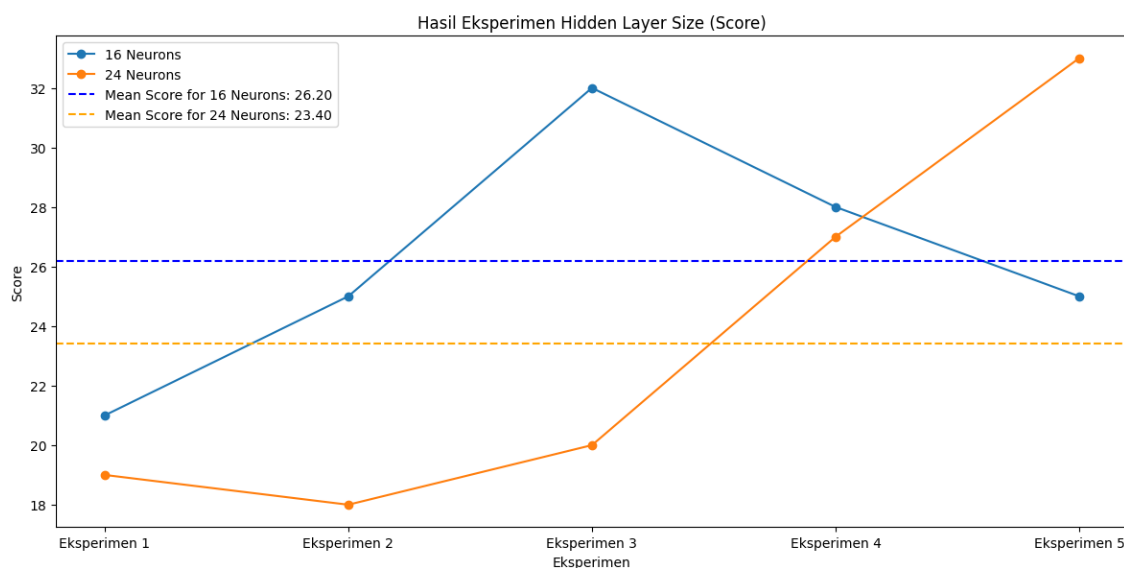
Semakin tinggi *mutation chance* pada sebuah operasi algoritma genetika, maka semakin tinggi juga perubahan matriks bobot yang akan terjadi. Untuk beberapa *case*, terutama pada penelitian ini *mutation chance* yang lebih tinggi dapat membuat ular dalam populasi

mengalami perubahan yang lebih besar. Hal ini dapat menyebabkan fluktuasi yang lebih besar dalam performa, mengganggu solusi yang sebelumnya sudah baik, dan menghambat proses penyempurnaan solusi. *Mutation chance* yang lebih tinggi dapat menyebabkan over-mutasi, di mana perubahan yang terlalu banyak dan terlalu sering mengakibatkan hilangnya solusi yang baik sebelum dapat dieksploitasi sepenuhnya. Hal ini dapat dibuktikan pada pengujian *mutation chance* yang menggunakan nilai 0.5%. Meskipun mendapatkan *score* yang cukup tinggi, namun terdapat 2 *score* yang bisa dibilang rendah, yaitu 14 pada eksperimen pertama dan 13 pada eksperimen ketiga, sehingga terjadinya inkonsistensi performa.

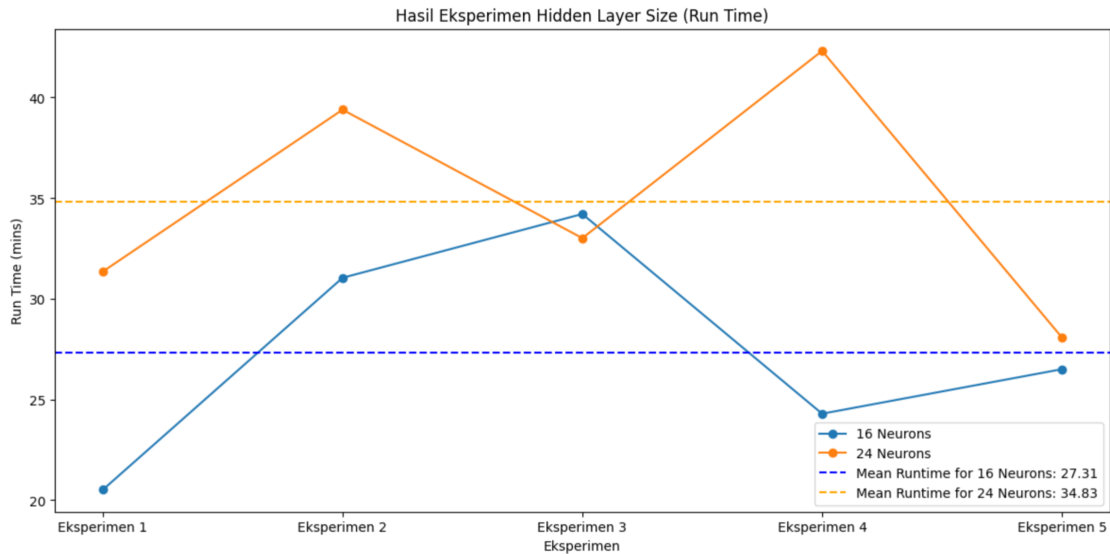
3.3 Perbandingan *Hidden layer* Size

Perbandingan *hidden layer* size atau jumlah *neuron* pada *hidden layer* merupakan pengujian terakhir pada penelitian ini. Semakin besar jumlah *neuron* yang digunakan maka semakin baik juga *neural networks* dalam menangkap pola dan interaksi yang ada pada permainan ular yang dapat meningkatkan performa dari ular, namun lebih banyak *neuron* berarti lebih banyak juga komputasi dari *hidden layer* sehingga *runtime* dari program meningkat. Perbandingan ini dilakukan untuk melihat apakah semakin banyak *neuron* dapat menciptakan perbedaan yang cukup signifikan dibandingkan dengan jumlah *neuron* yang lebih sedikit. Parameter *setting* yang digunakan untuk *hidden layer* 16 sama dengan parameter *setting* sebelumnya pada pengujian *mutation chance* 0.1%. Maka dari itu, hasil pengujian untuk *hidden layer* 16 akan diambil dari hasil pengujian *mutation chance* yang sebelumnya.

Hasil pengujian menunjukan bahwa *neuron* 16 memiliki keunggulan baik dari segi rata-rata *score* (Gambar 4.16) maupun rata-rata *runtime* yang lebih sedikit dibandingkan dengan *neuron* 24 (Gambar 4.17). Ukuran *hidden layer* yang lebih kecil mungkin lebih sesuai dengan kompleksitas pada permainan ular. Terlalu banyak *neurons* dapat menyebabkan *overfitting*, di mana model menjadi terlalu disesuaikan dengan data pelatihan dan tidak mampu menggeneralisasi dengan baik pada data yang baru. Selain itu, *runtime* yang lebih lama pada *neuron* 24 diakibatkan dari meningkatnya jumlah operasi komputasi yang diperlukan, sehingga memerlukan waktu yang lebih lama untuk menyelesaikannya. Hal ini lah yang membuat *neuron* 16 unggul baik dari rata-rata *score* maupun rata-rata *runtime*.

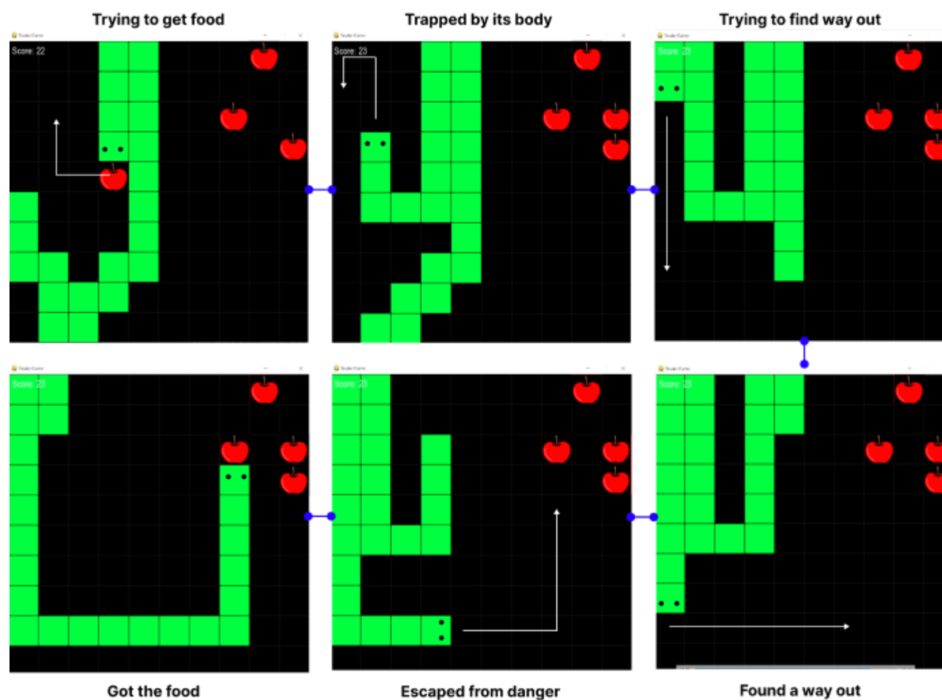


Gambar 20. Grafik *Score* pada Pengujian *Hidden Layer* Size



Gambar 21. Grafik *Runtime* pada Pengujian *Hidden Layer*

Dari sekian banyak eksperimen yang telah dilakukan, terdapat salah satu eksperimen yang menunjukan bagaimana algoritma genetika dan *neural networks* dapat membuat brain ular ini menjadi benar-benar cerdas. Pada Gambar 22 terlihat bagaimana ketika ular tersebut terjebak oleh badannya sendiri dan bagaimana cara ular tersebut keluar dari bahayanya. *Neural networks* membantu dalam pengambilan keputusan, sementara algoritma genetika mengoptimalkan solusi-solusi yang dihasilkan dari *neural networks*, sehingga ular menunjukkan pergerakan yang strategis dan aman, menghindari dirinya sendiri dan mencari jalur terpendek ke makanan, menandakan keberhasilan dalam penerapan dan pelatihan algoritma. Inilah alasan kenapa sangat penting adanya GUI pada program yang dibuat, apa yang terjadi pada Gambar 22 dapat dijadikan sebuah acuan terkait berhasilnya penerapan algoritma genetika dan *neural networks*.



Gambar 22. Tidak Hanya Mendapatkan Makanan, tapi Bisa Menghindari Bahaya

4. KESIMPULAN

Berdasarkan hasil pengujian, dapat disimpulkan bahwa konfigurasi parameter dengan generasi lebih besar dari populasi adalah yang paling optimal. Konfigurasi ini menghasilkan skor yang hampir setara dengan generasi dan populasi sama besar, namun dengan runtime yang jauh lebih efisien. Pengujian *mutation chance* menunjukkan bahwa nilai 0.1% adalah yang terbaik, memberikan stabilitas evolusi dan performa optimal tanpa gangguan dari mutasi yang berlebihan. Untuk ukuran *hidden layer*, 16 *neuron* terbukti lebih efisien dibandingkan 24 *neuron*, baik dari segi skor maupun runtime. Secara keseluruhan, konfigurasi dengan generasi lebih besar dari populasi, *mutation chance* 0.1%, dan *hidden layer* 16 *neuron* menawarkan keseimbangan terbaik antara kinerja dan efisiensi waktu dalam penerapan algoritma genetika dan *feed-forward neural networks* pada permainan ular. Kombinasi algoritma genetika dan *neural networks* memerlukan banyak iterasi dan generasi untuk menemukan solusi optimal. Hal ini menyebabkan waktu komputasi yang tinggi, terutama pada populasi dan jumlah generasi yang besar. Selain itu, penentuan parameter yang optimal (seperti *mutation rate*, ukuran populasi, jumlah generasi, dan arsitektur *neural networks*) memerlukan banyak eksperimen dan dapat menjadi sangat kompleks untuk mendapatkan performa yang maksimal. Maka dari itu, untuk penelitian yang selanjutnya mungkin bisa menerapkan teknik optimasi seperti penggunaan distributed computing atau parallel processing untuk mempercepat proses pelatihan, serta dapat menggunakan algoritma hyperparameter tuning otomatis, seperti Bayesian Optimization atau Grid Search, untuk menemukan parameter yang optimal dengan lebih efisien.

DAFTAR RUJUKAN

B. HalmosiC. Sik-Lányi. (2019). *Learning to play snake using genetic neural networks*. https://www.researchgate.net/publication/343281468

Białas, P. (2019). *Implementation of artificial intelligence in Snake game using genetic algorithm and neural networks*. https://ceur-ws.org/Vol-2468/p9.pdf

Boris, T., & Goran, S. (2017). Evolving neural network to play game 2048. *24th Telecommunications Forum, TELFOR 2016*. https://doi.org/10.1109/TELFOR.2016.7818911

Brown, J. A., de Araujo, L. J. P., & Grichshenko, A. (2021). *Snakes AI Competition 2020 and 2021 Report*. http://arxiv.org/abs/2108.05136

Carr, J. (2014). *An Introduction to Genetic Algorithms*. Senior Project (Vol. 1, no. 40, pp. 7).

Chi Yuen, M., Wuan Yeong, L., Chen Yi Kang, E., Qaisar Syed, S., & Arabee Abdul Salam, Z. (2021). Investigating parameters of genetic algorithm and neural network on classic snake game. *Journal of Applied Technology and Innovation*, *5*(2).

Hamdia, K. M., Zhuang, X., & Rabczuk, T. (2021). An efficient optimization approach for designing machine learning models based on genetic algorithm. *Neural Computing and Applications*, *33*(6), 1923–1933. https://doi.org/10.1007/s00521-020-05035-x

Hau Hor, S., Jeh Tan, S., Kye Yan, M., Arabee bin Abdul Salam, Z., & Shin Sim, Y. (2022). Snake Game: A genetic neural network approach. In *Journal of Applied Technology and Innovation, 6*(1).

Kong, S., & Mayans, J. A. (2021). *Automated Snake Game Solvers via AI Search Algorithms*.

Kumar, R., Memoria, M., Gupta, A., & Awasthi, M. (2021). Critical Analysis of Genetic Algorithm under Crossover and Mutation Rate. *Proceedings - 2021 3rd International Conference on Advances in Computing, Communication Control and Networking, ICAC3N 2021*, (pp. 976–980). https://doi.org/10.1109/ICAC3N53548.2021.9725640

Ma, Y. (2024). Optimization of basic PID control algorithm based on genetic algorithm and Matlab. *Theoretical and Natural Science*, *30*(1), 178–186. https://doi.org/10.54254/2753-8818/30/20241103

Miller, M., Washburn, M., & Khosmood, F. (2019). Evolving unsupervised neural networks for Slither.io. *ACM International Conference Proceeding Series*. https://doi.org/10.1145/3337722.3341837

Mishra, Y., Kumawat, V., & Selvakumar, K. (2019). Performance analysis of flappy bird playing agent using neural network and genetic algorithm. *Communications in Computer and Information Science*, *1025 CCIS*, (pp. 253–265). https://doi.org/10.1007/978-981-15-1384-8_21

Rahul Ramesh Patil. (2023). AI-Infused Algorithmic Trading: Genetic Algorithms and Machine Learning in High-Frequency Trading. *International Journal For Multidisciplinary Research*, *5*(5). https://doi.org/10.36948/IJFMR.2023.V05I05.5752

Shiruru, K. (2016). *An Introduction To Artificial Neural Network*. https://www.researchgate.net/publication/319903816

Uthansakul, P., Anchuen, P., Uthansakul, M., & Khan, A. A. (2020). QoE-Aware Self-Tuning of Service Priority Factor for Resource Allocation Optimization in LTE Networks. *IEEE Transactions on Vehicular Technology*, *69*(1), 887–900. https://doi.org/10.1109/TVT.2019.2952568

Wei, Z., Wang, D., Zhang, M., Tan, A. H., Miao, C., & Zhou, Y. (2018). Autonomous agents in snake game via deep reinforcement learning. *Proceedings - 2018 IEEE International Conference on Agents, ICA 2018*, (pp. 20–25). https://doi.org/10.1109/AGENTS.2018.8460004

Zhang, R., & Cai, R. (2020). *Train a snake with reinforcement learning algorithms*. https://openreview.net/forum?id=iu2XOJ45cxo