حسین لشگری – 9731055 گزارش پروژه دوم

## نحوه مدل سازی :

- از کلاس **Game** که دربردارنده سطح بازی و تابع شایستگی که برای سنجش شایستگی کروموزوم ها است، استفاده می کنیم.
- از کلاس **Chromosome** که دربردارنده رشته، برنده یا بازنده بودن و امتیاز کروموزوم است، استفاده می کنیم.
- از کلاس **Generation** که دربردارنده اعضای یک نسل، بهترین و بدترین کروموزوم، میانگین امتیاز کروموزوم ها و یک **flag** که برای بررسی وجود یا عدم وجود کروموزوم برنده در نسل است، استفاده می کنیم.

طبق مراحل 5 گانه ی تولید نسل در الگوریتم ژنتیک به تولید نسل می پردازیم.

## توضیح نحوه پیاده سازی مراحل 5 گانه الگوریتم ژنتیک :

### 1. تولید نسل اولیه:

برای اینکار از تابع **generate_init_population** استفاده می کنیم. طبق کد زیر این تابع تعداد اعضای نسل اولیه، بازی موردنظر و حالت امتیازدهی در تابع شایستگی را می گیرد و به تعداد مورد نظر کروموزوم با طول مطابق بازی و به احتمال رندوم که احتمال 0 از 1 و 2 بیشتر است، تولید می کند.

سپس شایستگی کروموزوم ها را توسط تابع شایستگی بررسی می کند و نسل اولیه را برمی گرداند.

```
def generate_init_population(pop_number: int, game: Game, score_mode: int = 0):
    pop = []
    for ch in range(0, pop_number):
        temp_str = ""
        for st in range(0, game.level_len):
            rand = random.random()
            if rand > 0.4:
                temp_str += '0'
            elif rand > 0.2:
                temp_str += '1'
            else:
                temp_str += '2'
        flag, score = game.get_score(temp_str, score_mode)
        pop.append(Chromosome(copy.deepcopy(temp_str), copy.deepcopy(flag), copy.deepcopy(score)))
    init_gen = Generation(pop)
    return init_gen
```

## 2. محاسبه شایستگی:

برای محاسبه شایستگی کروموزوم ها از تابع **get_score** موجود در کلاس **Game** استفاده می کنیم.

این تابع رشته کروموزوم و حالت امتیازدهی را می گیرد و کاراکتر به کاراکتر روی رشته حرکت می کند و امتیازدهی را مطابق آنچه در دستورکار خواسته شده اعمال می کند. به این صورت که:

- امتیازی که در کل یک کروموزوم می گیرد متناسب با بیشترین طولی از رشته که کروموزوم به سلامت مسیر را طی می کند و نمی سوزد است. برای اینکار از دو متغیر **max_dis** و **current_dis** که برای ذخیره تعداد کاراکتر های درست در رشته است استفاده می کنیم. متغیر **current_dis** طول تک تک مسیرهای درست را ذخیره می کند و اگر از مقدار **max_dis** بیشتر بود با آن جایگزین می کند.
- مطابق کد تمام قسمت های امتیازی برای کروموزوم پیاده سازی شدند و در انتها با توجه به **mode** که برای امتیازدهی داریم یا امتیاز برنده شدن محاسبه می شود یا خیر.
نحوه امتیازدهی به این صورت است که اگر یک کروموزوم تمام قسمت های امتیازی را بگیرد آنگاه دو برابر **max_dis** به عنوان امتیاز برگردانده می شود. یعنی قسمت های امتیازی درنهایت نسبتی بین 0 تا **max_dis** امتیاز می دهند و بیشتر از آن نمی دهند. اینکار به منظور اهمیت بیشتر طول درست کروموزوم نسبت به بقیه موارد صورت گرفته است.

## 3. انتخاب:

مطابق کد زیر برای انتخاب بهترین های یک نسل، دو **mode** مختلف پیاده سازی شده اند.

حالت اول اینکه نصف کروموزوم های با بیشترین امتیاز انتخاب می شوند و برگردانده می شوند.

حالت دوم اینکه از **3/4** آرایه کروموزوم ها که دارای بیشترین امتیاز هستند، به طور وزن دار و تصادفی با جایگذاری، به اندازه نصف جمعیت اولیه انتخاب می شوند و برگردانده می شوند.

برای اینکار از یک آرایه که به نسبت امتیاز هر کروموزم آن ها به آرایه اضافه می شوند استفاده می شود و درنهایت به طور تصادفی از این آرایه تعدادی انتخاب می شوند.

```
def selection(gen: Generation, mode: int = 0):
    selected = []
    if mode == 0:
        gen.members.sort(key=lambda x: x.score, reverse=True)
        selected = copy.deepcopy(gen.members[0: int(gen.number/2)])
    elif mode == 1:
        gen.members.sort(key=lambda x: x.score, reverse=True)
        min_score = gen.members[int(3 * gen.number / 4)].score
        chance_arr = []
        for ch in range(0, int(3 * gen.number / 4)):
            for j in range(0, int(gen.members[ch].score - min_score) + 1):
                chance_arr.append(ch)
        for b in range(0, int(gen.number / 2)):
            selected.append(copy.deepcopy(gen.members[random.choice(chance_arr)]))
    return selected
```

## 4. بازترکیبی:

مطابق کد پایین، برای بازترکیبی فرزندان **mode** 3 مختلف درنظر گرفته شده است.

در هر حالت ابتدا دو تا از والدین به صورت تصادفی انتخاب می شوند.

در حالت اول، کاراکترهای فرزند اول به صورت رندوم از یکی از دو والد انتخاب می شود و برای فرزند دیگر، از والدی که انتخاب نشده بود انتخاب می شود.

در حالت دوم، به صورت بازترکیبی یک نقطه ای، هر والد به دو قسمت مساوی تقسیم شده و هر بخش به صورت ضربدری در یک فرزند استفاده می شود.

در حالت سوم، به صورت بازترکیبی دونقطه ای هر والد به 3 قسمت مساوی تقسیم می شود و هر دوبخش نامجاور از هر والد به همراه قسمت وسط از والد دیگر یک فرزند را تشکیل می دهند.

```
def crossover(selected: list, mode: int = 0):
    pop1 = copy.deepcopy(selected)
    pop2 = copy.deepcopy(selected)
    children = []
    parents = [None, None]
    for p in range(0, len(selected)):
        child1_str = ""
        child2_str = ""
        parents[0] = pop1.pop(random.randint(0, len(pop1)-1))
        parents[1] = pop2.pop(random.randint(0, len(pop2)-1))
        if mode == 0:
            for j in range(0, parents[0].len):
                choose = random.randint(0, 1)
                child1_str += parents[choose].string[j]
                child2_str += parents[1 - choose].string[j]
        elif mode == 1:
            child1_str += parents[0].string[0: int(parents[0].len / 2)]
            child1_str += parents[1].string[int(parents[1].len / 2): parents[1].len]
            child2_str += parents[1].string[0: int(parents[1].len / 2)]
            child2_str += parents[0].string[int(parents[0].len / 2): parents[0].len]
        elif mode == 2:
            child1_str += parents[0].string[0: int(parents[0].len / 3)]
            child1_str += parents[1].string[int(parents[1].len / 3): int(2 * parents[1].len / 3)]
            child1_str += parents[0].string[int(2 * parents[0].len / 3): parents[0].len]
            child2_str += parents[1].string[0: int(parents[1].len / 3)]
            child2_str += parents[0].string[int(parents[0].len / 3): int(2 * parents[0].len / 3)]
            child2_str += parents[1].string[int(2 * parents[1].len / 3): parents[1].len]
        children.append(Chromosome(copy.deepcopy(child1_str)))
        children.append(Chromosome(copy.deepcopy(child2_str)))
    return children
```

## 5. جهش:

مرحله آخر از این مراحل، جهش است. برای اینکار از تابع mutation استفاده می کنیم. این تابع لیست فرزندان و احتمال جهش را می گیرد و به صورت تصادفی تعدادی از کاراکترهای هر فرزند را انتخاب می کند و با احتمال گرفته شده آن را به 0 ، 1 یا 2 تغییر می دهد.

تعداد کاراکترهای انتخاب شده با توجه به احتمال می تواند 1/15 یا 1/10 یا 1/5 طول رشته باشد که احتمال یکی از 15 کاراکتر بیشتر از 5 و 10 کاراکتر است.

همچنین احتمال اینکه 0 به عنوان کاراکتر جایگزین شده انتخاب شود بسیار بیشتر از 1 و 2 است.

```
def mutation(children: list, p: float = 0.2):
    for ch in children:
        num = random.random()
        if num > 0.5:
            num = int(ch.len / 15)
        elif num > 0.2:
            num = int(ch.len / 10)
        else:
            num = int(ch.len / 5)

        which = random.sample(range(0, ch.len), num)
        temp_list = list(ch.string)
        for act in which:
            if random.random() < p:
                p = random.random()
                if p > 0.2:
                    m = '0'
                elif p > 0.1:
                    m = '1'
                else:
                    m = '2'
                temp_list[act] = m
        ch.string = copy.deepcopy("".join(temp_list))
    return children
```

## الگوریتم ژنتیک :

این الگوریتم از توابع پیاده سازی شده که بیان شدند استفاده می کند و مراحل مربوطه را در یک حلقه به طور پیوسته ادامه می دهد.

این الگوریتم، بازی، تعداد جمعیت هر نسل، حالت selection، حالت crossover، حالت امتیازدهی و احتمال جهش را می گیرد.

ابتدا نسل اولیه تولید می شود و به آرایه نسل ها اضافه می شود سپس در یک حلقه که حداکثر تا 500 نسل دیگر ادامه می یابد، ابتدا از هر نسل تعدادی انتخاب می شوند سپس توسط بازترکیبی و جهش، نسل جدید به وجود می آید که امتیاز هر کدام از کروموزوم ها توسط تابع get_score دریافت می شود.

سپس برای خاتمه دادن به تولید نسل، همگرایی نسل ها بررسی می شوند. برای اینکار امتیاز بهترین کروموزوم از نسل قبلی در prev_best ذخیره می شود و با امتیاز بهترین کروموزوم در نسل حال حاضر مقایسه می شود.

اگر به تعداد 30 نسل پشت سر هم اختلاف این دوعدد کمتر از یک درصد باشد آن گاه دچار همگرایی شده ایم و کار الگوریتم به پایان می رسد و آرایه نسل ها برگردانده می شود.

```
def genetic_algorithm(game: Game, pop_number: int, s_mode: int = 0, c_mode: int = 0, score_mode: int = 0,
                      mutation_p: float = 0.2):
    init_gen = generate_init_population(pop_number, game, score_mode)
    generations = [init_gen]
    next_gen = init_gen
    prev_bst = -1000
    j = 0
    for gen in range(0, 500):
        selected_pop = selection(next_gen, s_mode)
        children_pop = crossover(selected_pop, c_mode)
        mutant_pop = mutation(children_pop, mutation_p)
        for ch in mutant_pop:
            ch.flag, ch.score = game.get_score(ch.string, score_mode)
        next_gen = Generation(mutant_pop)
        generations.append(next_gen)
        if 0.99 < next_gen.best.score/prev_bst < 1.01:
            j += 1
            if j == 30:
                return generations
        else:
            j = 0
        prev_bst = next_gen.best.score
    return generations
```

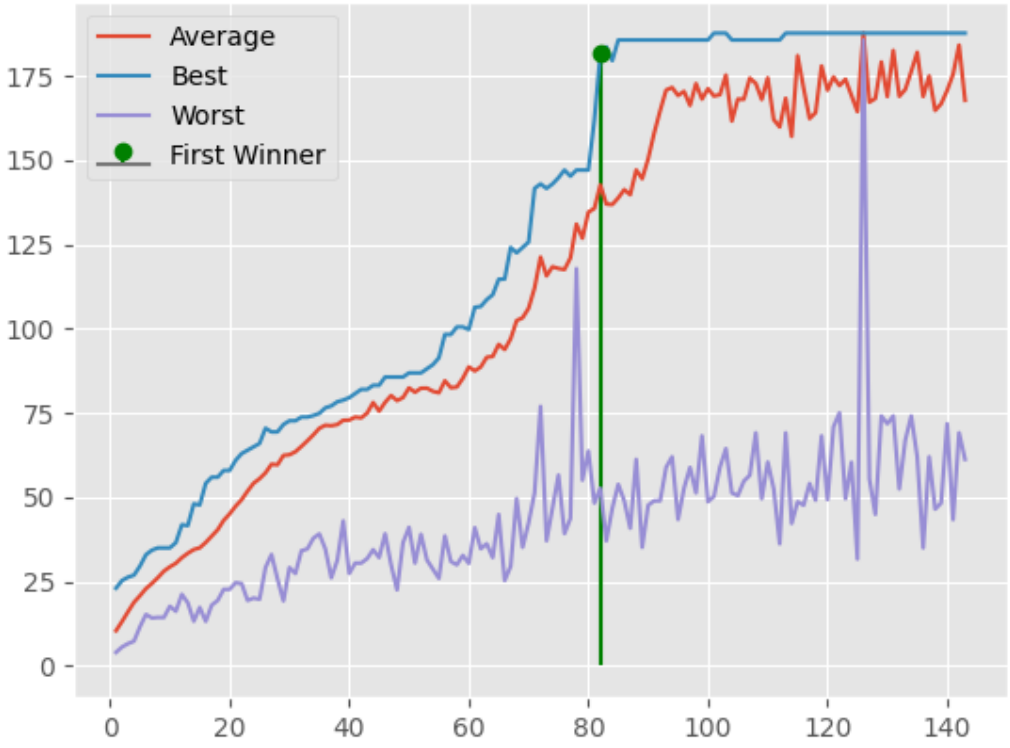
## نمودارها و مقایسه :

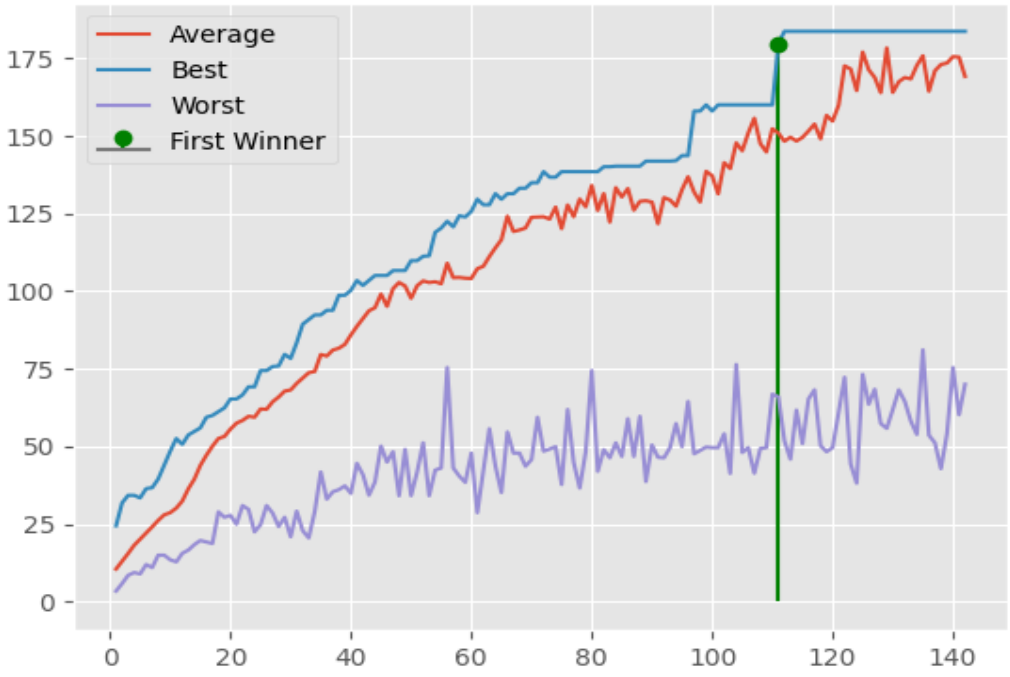
از آنجایی که احتمال در تعیین زمان همگرایی، میانگین، بهترین و بدترین و اولین نسل برنده نقش بسزایی دارد، از هر روش اجرای الگوریتم، 5 مورد اجرا می کنیم و میانگین آن ها را به عنوان نتیجه کلی درنظر می گیریم.

- اولین روش اجرا با در نظر گرفتن 200 کروموزوم و امتیاز برنده شدن به همراه انتخاب بهترین ها و بازترکیبی یک نقطه ای با احتمال جهش 0.1 است.
- روش دوم با در نظر گرفتن 500 کروموزوم و بدون محاسبه امتیاز برنده شدن به همراه انتخاب وزن دار براساس شایستگی و بازترکیبی دونقطه ای با احتمال جهش 0.5 است.

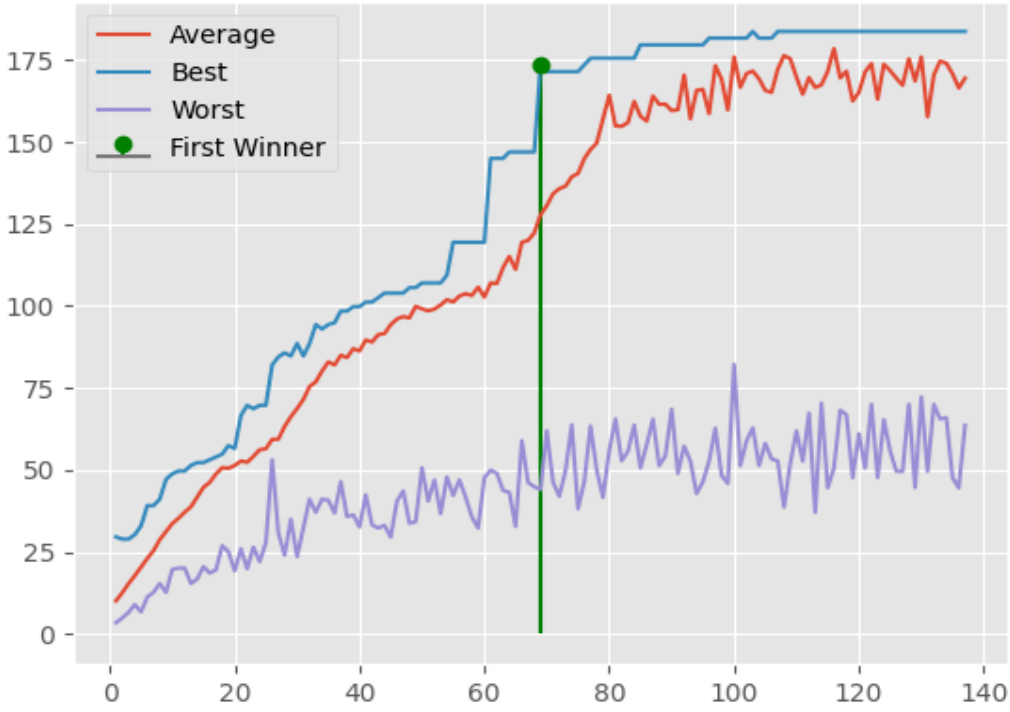
### روش اول:

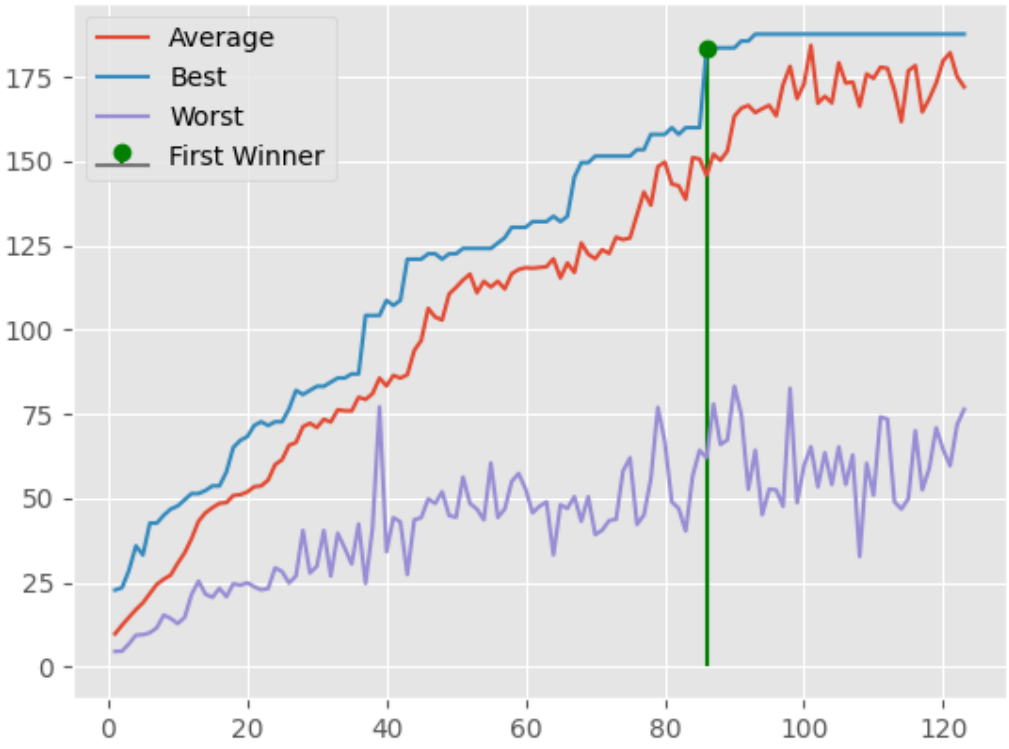
5 نمودار مربوط به اجرای این الگوریتم در سطح شماره 10 بازی، با روش اول در زیر مشاهده می شود:

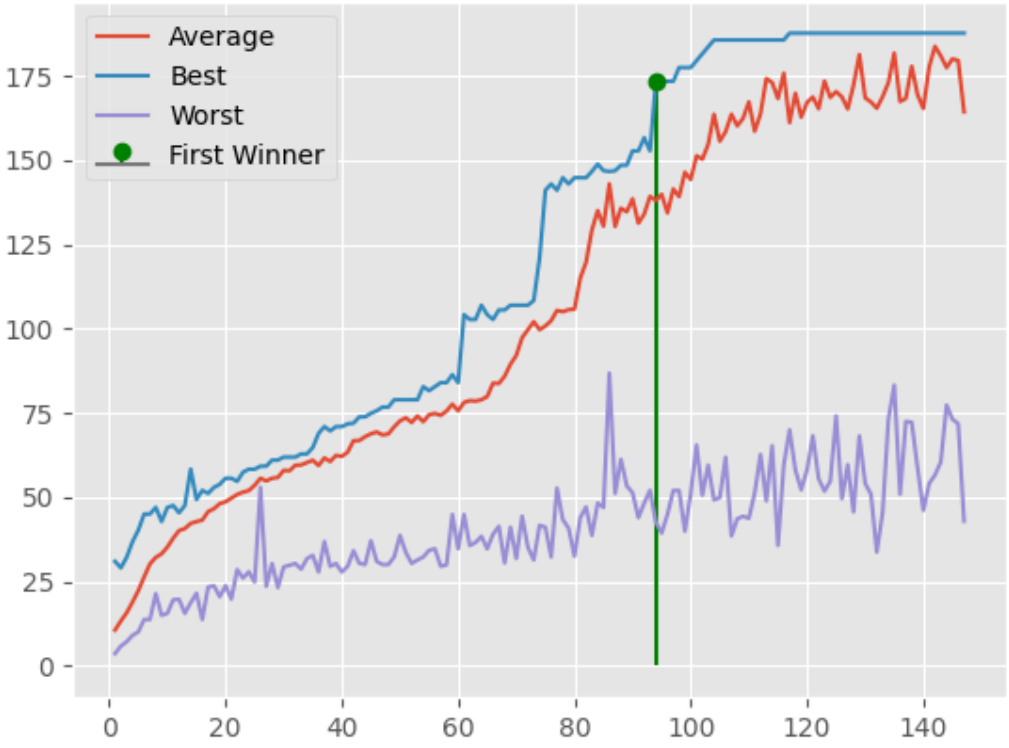





همانطور که در نمودار بالا مشاهده می شود الگوریتم دو بار در شانه گیر می کند که با توجه به جهشی که اتفاق افتاده از شانه رها شده و برنده می شود.

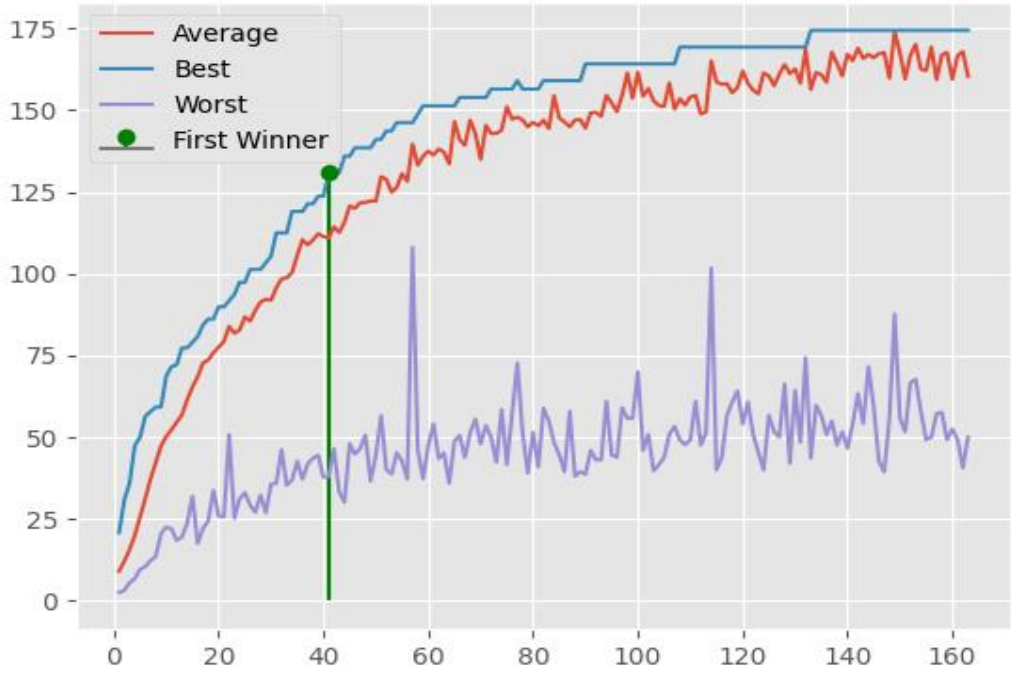


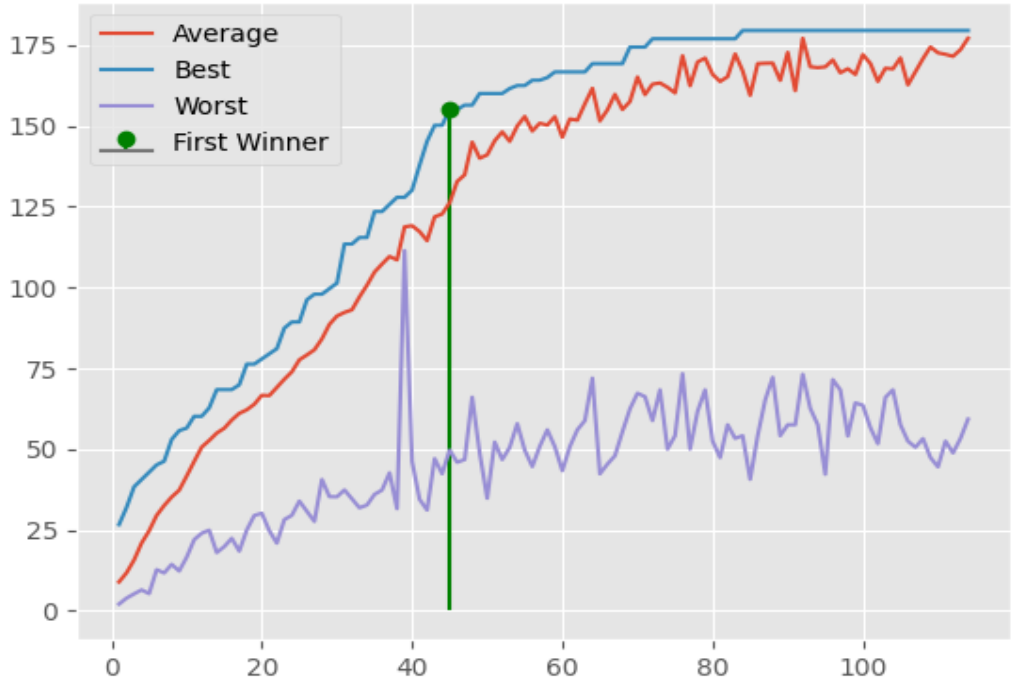

Average
Best
Worst
First Winner
175
150
125
100
75
50
25
0
0
20
40
60
80
100
120

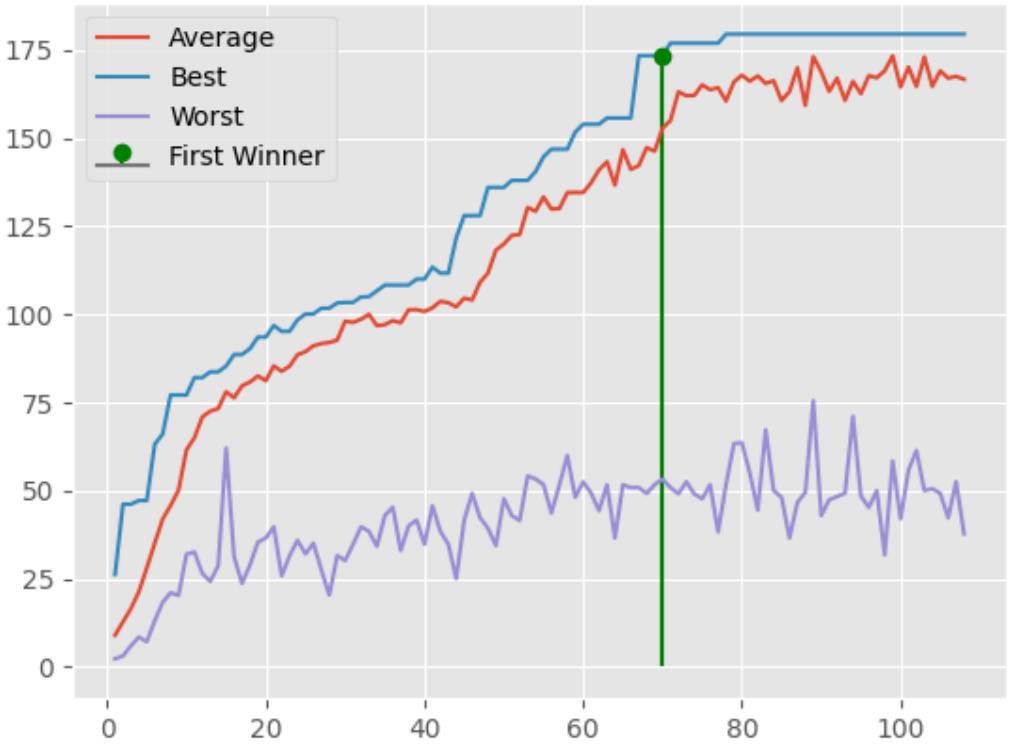

Average
Best
Worst
First Winner
175
150
125
100
75
50
25
0
0
20
40
60
80
100
120
140

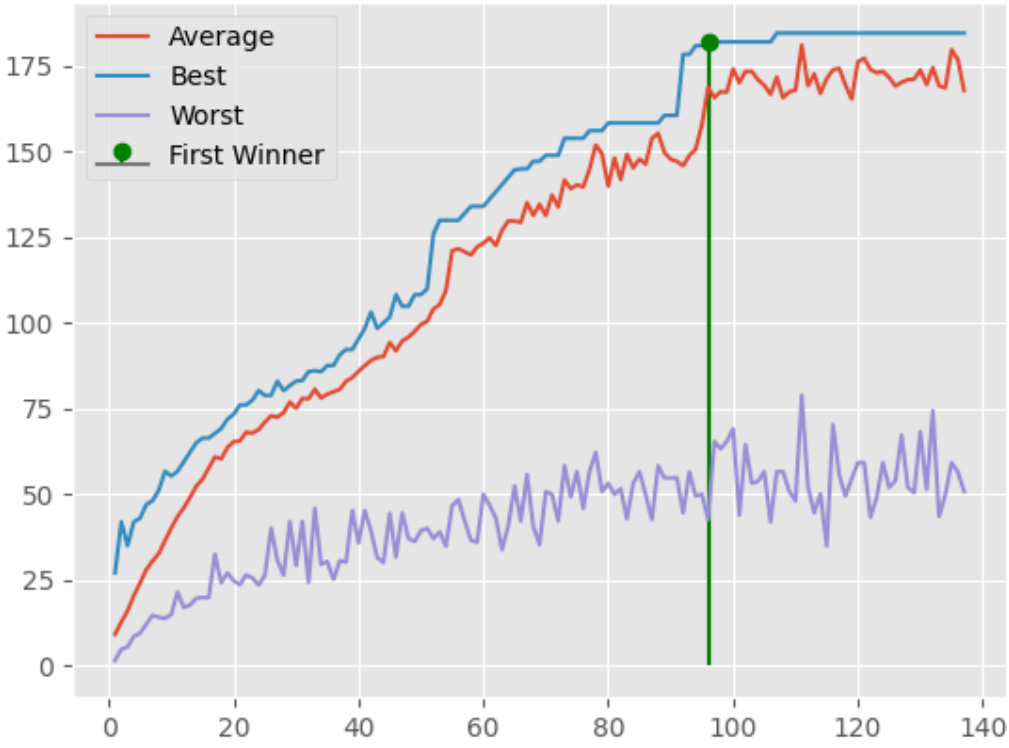
روش دوم:

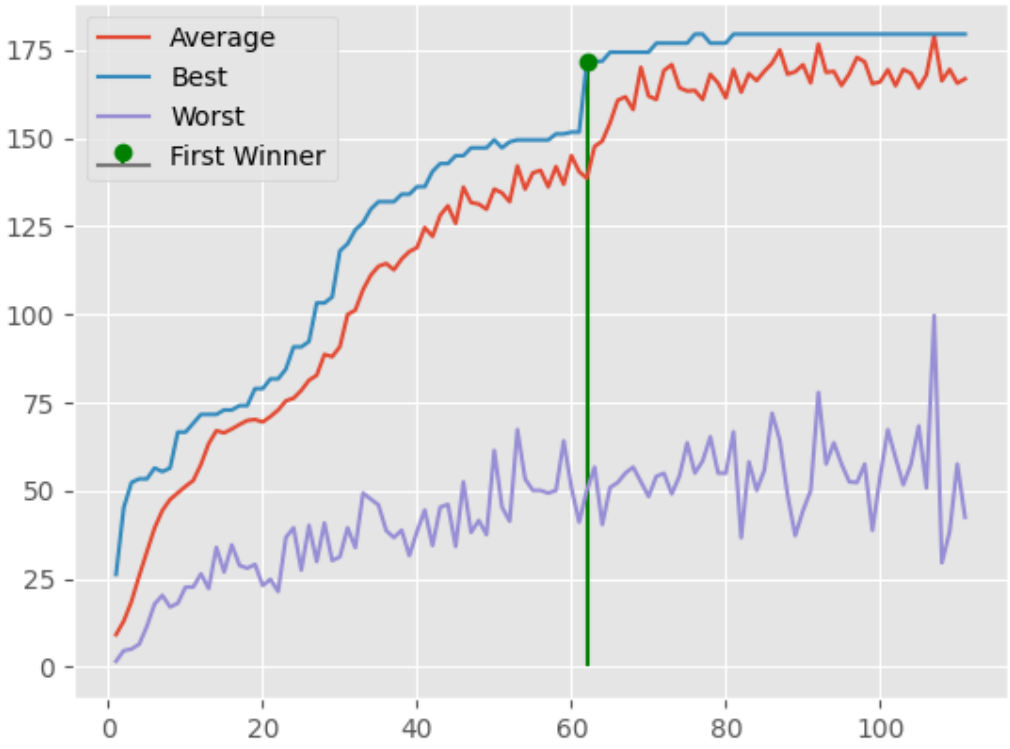
5 نمودار مربوط به اجرای این الگوریتم در سطح شماره 10 بازی، با روش دوم در زیر مشاهده می شود:




Average
Best
Worst
First Winner
175
150
125
100
75
50
25
0
0
20
40
60
80
100

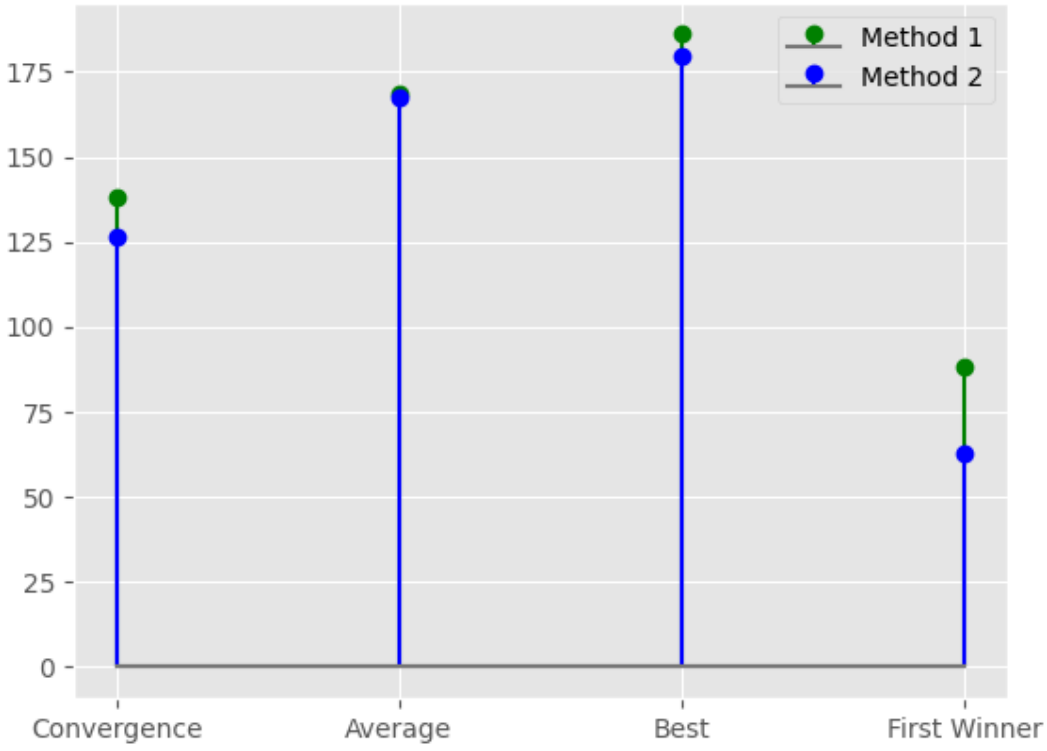

Average
Best
Worst
First Winner
175
150
125
100
75
50
25
0
0
20
40
60
80
100
120
140


Average
Best
Worst
First Winner
175
150
125
100
75
50
25
0
0
20
40
60
80
100

## مقایسه دو روش با توجه به نتایج بالا:

- همانطور که در شکل پایین می بینیم، روش اول به طور میانگین به تعداد نسل بیشتری نیاز دارد تا به همگرایی برسد(140). درحالی که روش دوم تقریبا با تولید 125 نسل به همگرایی می رسد.
- امتیاز میانگین نسل ها در هر دو روش با هم برابر است، درحالی که امتیاز بهترین کروموزوم در روش اول از روش دوم کمی بیشتر است.
- در روش اول با تولید میانگین90 نسل به یک راه حل می رسیم در حالی که در روش دوم با تولید 60 نسل به راه حل می رسیم.

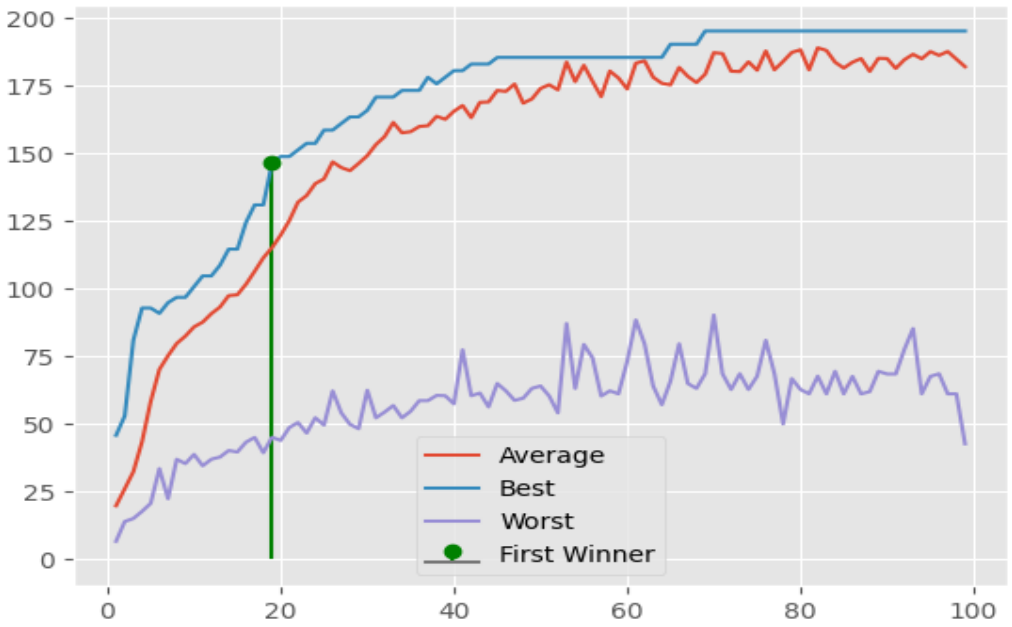
در کل با بررسی نتایج می بینیم روش دوم به نتیجه بهتری از روش اول می رسد.

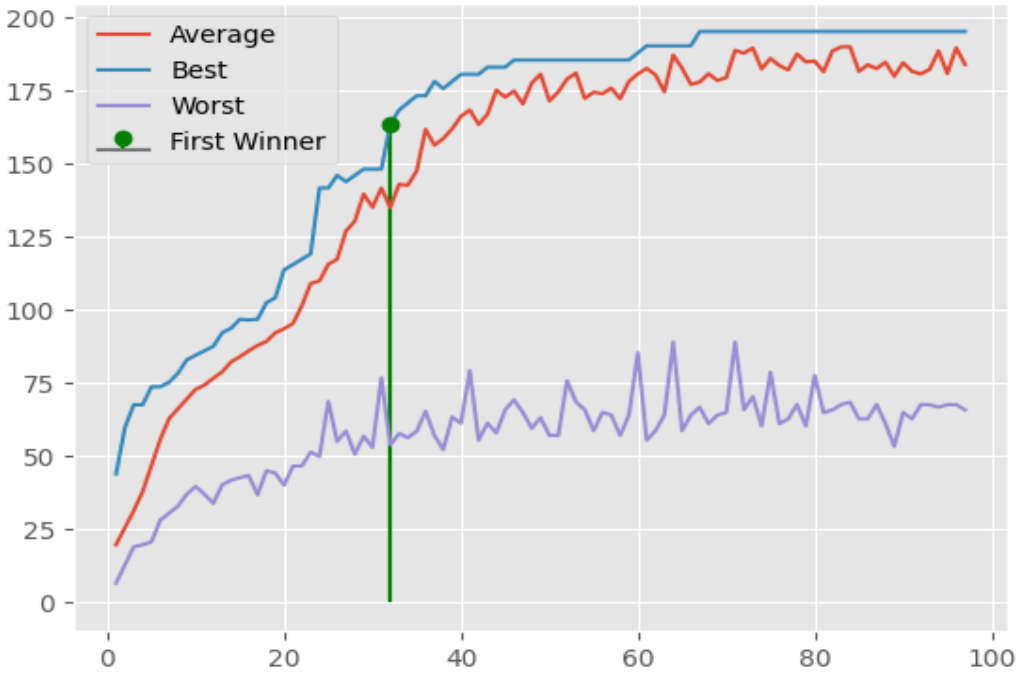


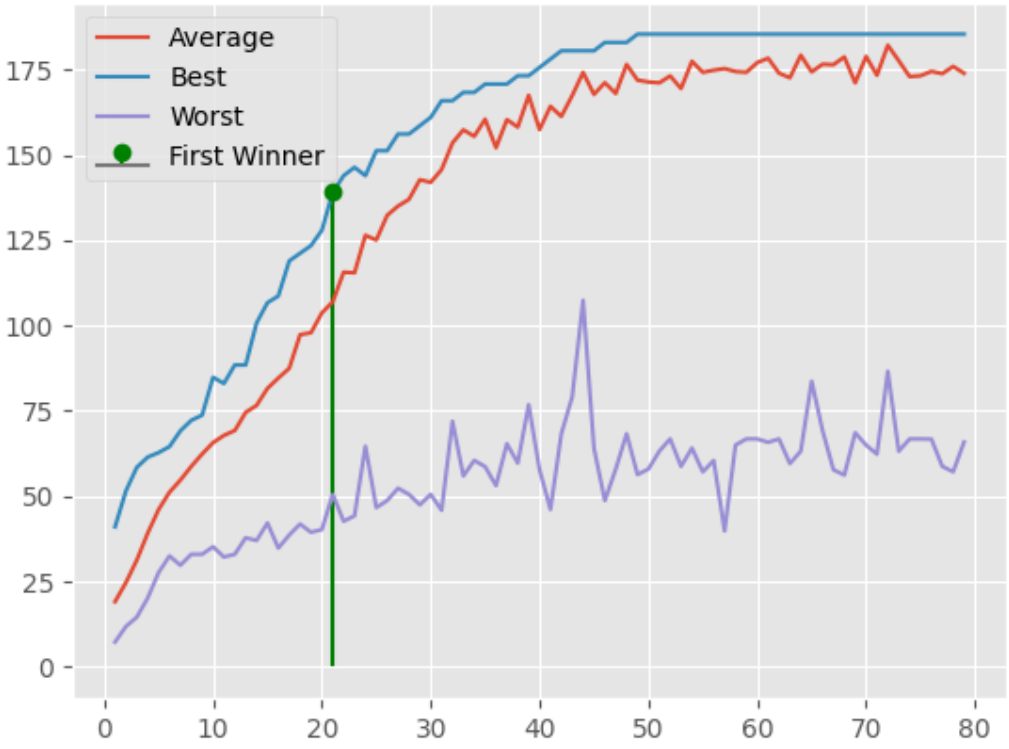
## مقایسه سرعت همگرایی در دو مرحله سخت و آسان :

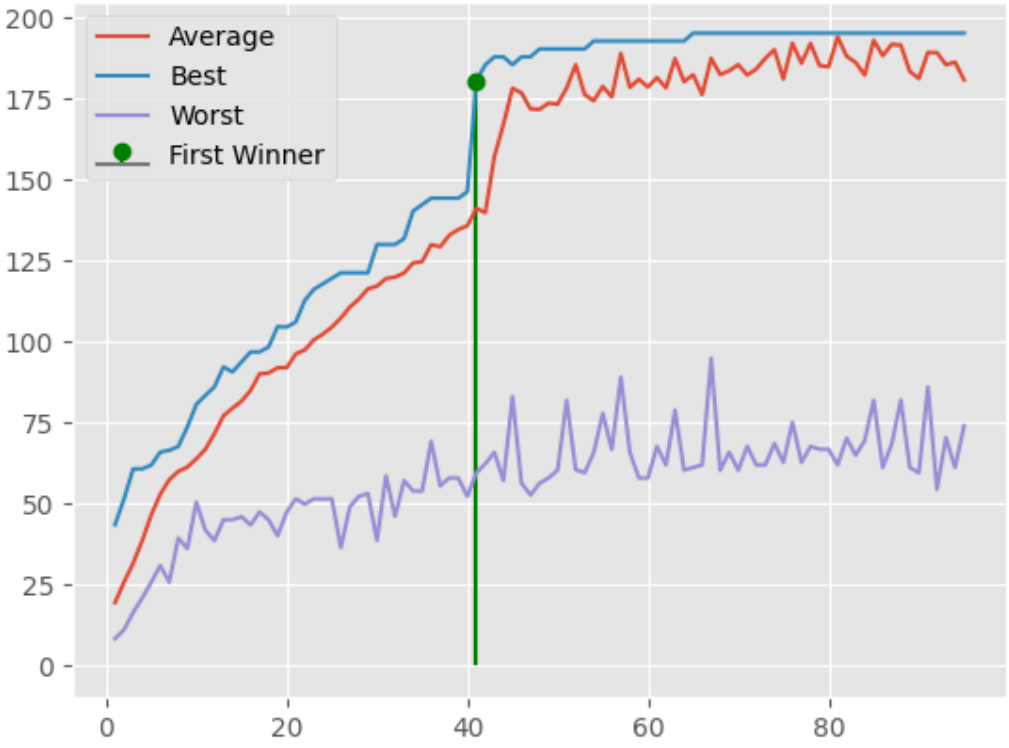
برای مقایسه، از دو سطح 6 و 9 استفاده می کنیم.

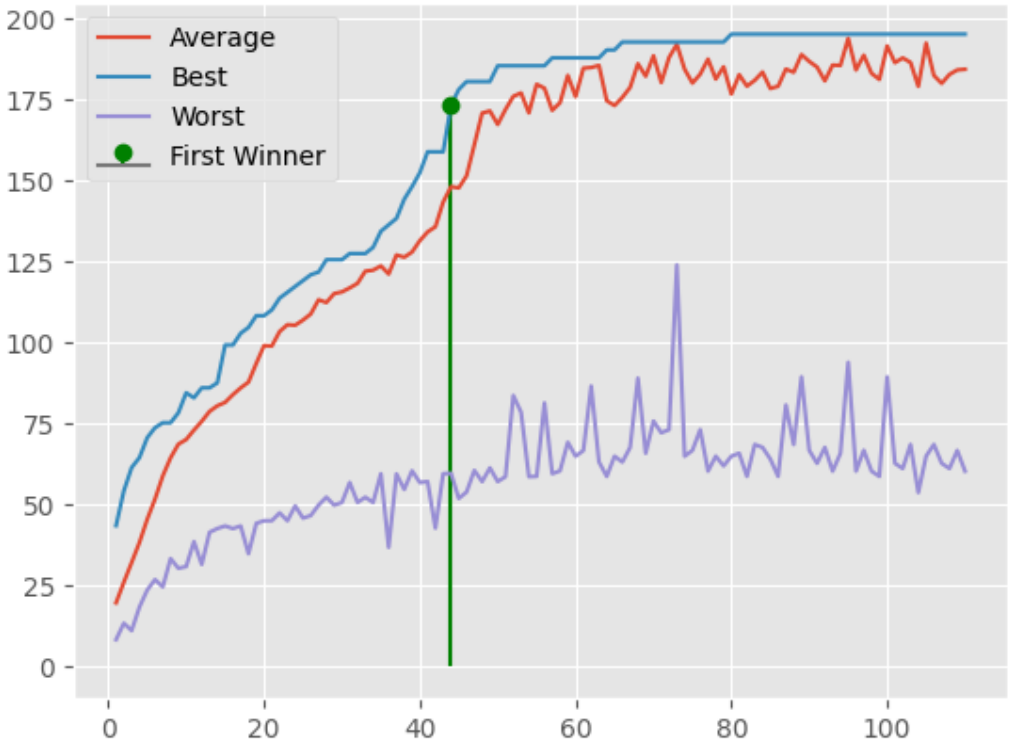
5 مورد از اجرای سطح 9 را در زیر می بینیم :



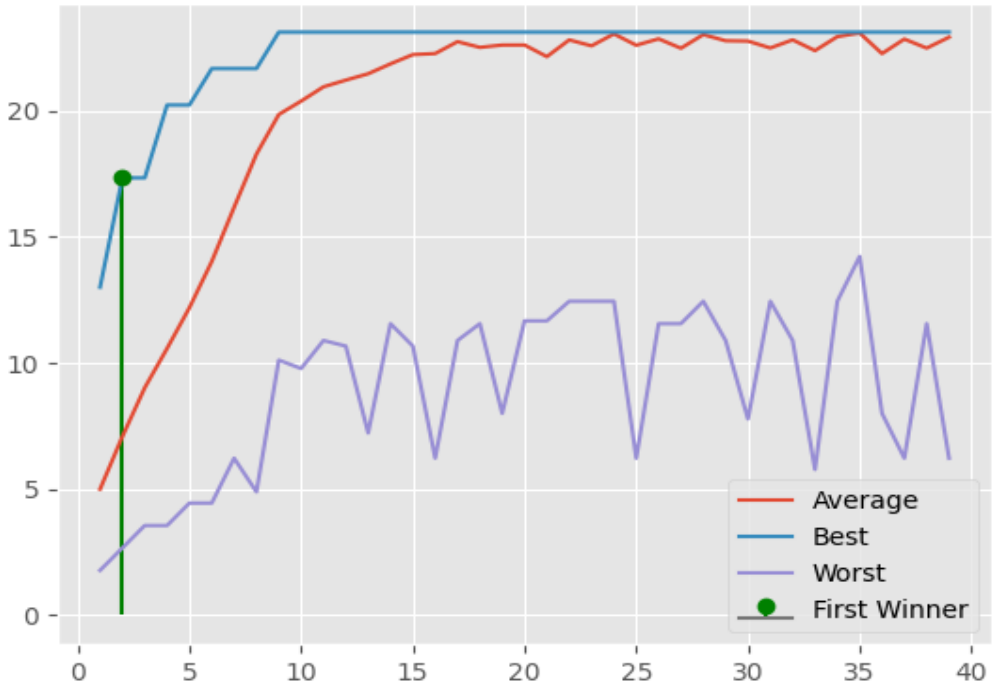



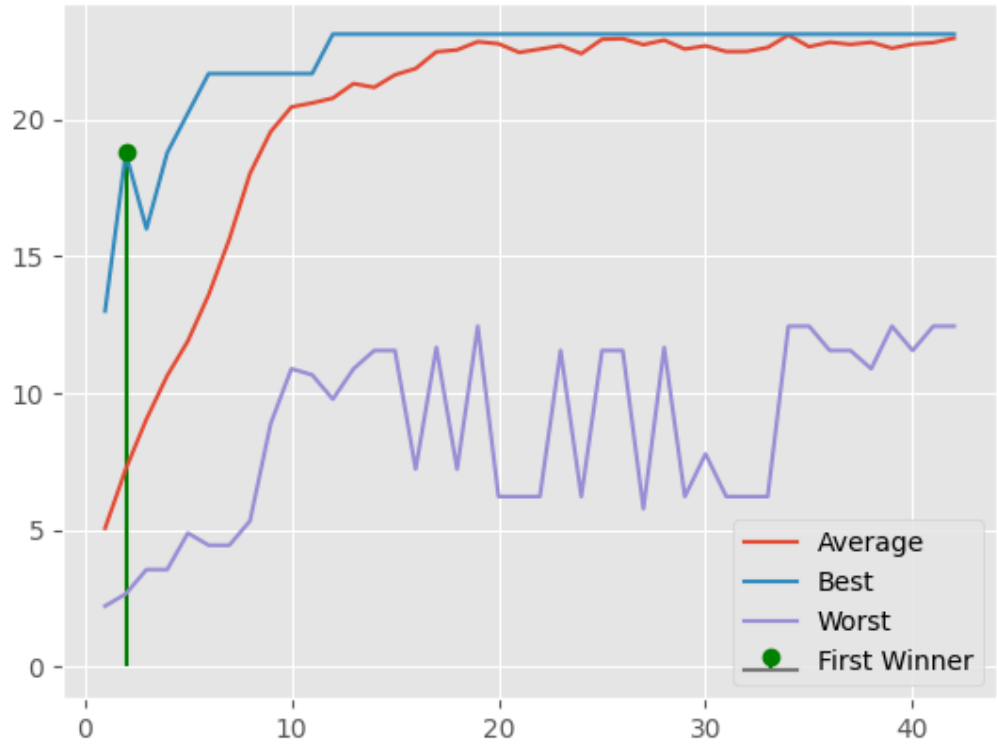

Average
Best
Worst
First Winner
175
150
125
100
75
50
25
0
0
10
20
30
40
50
60
70
80

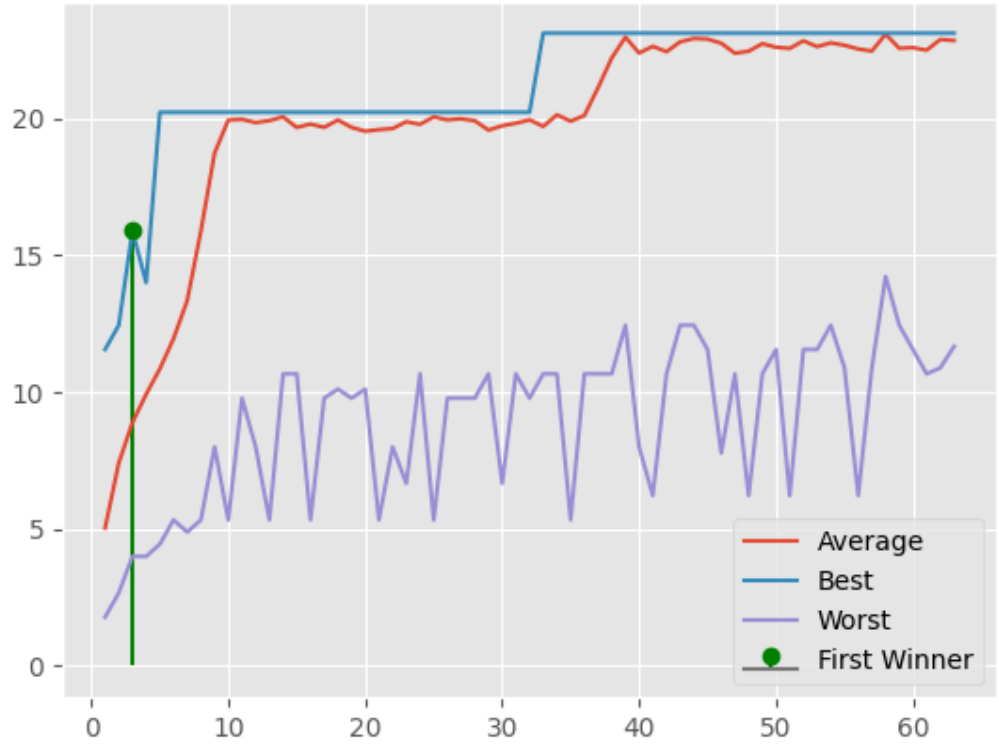

Average
Best
Worst
First Winner
200
175
150
125
100
75
50
25
0
0
20
40
60
80



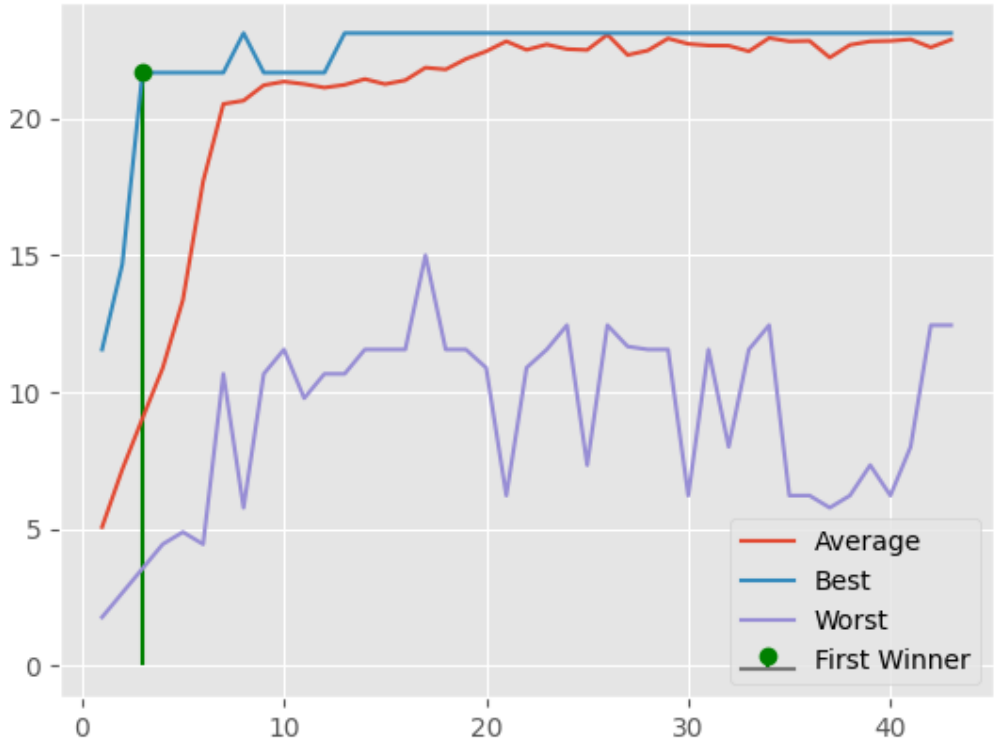

5 مورد از اجرای سطح 6 را در زیر می بینیم :




20
15
10
5
0
0
10
20
30
40
Average
Best
Worst
First Winner


20
15
10
5
0
0
10
20
30
40
50
60
Average
Best
Worst
First Winner


20
15
10
5
0
0
5
10
15
20
25
30
Average
Best
Worst
First Winner


20
15
10
5
0
0
10
20
30
40
Average
Best
Worst
First Winner

با مقایسه نتایج بالا می بینیم که میانگین تعداد تولید نسل تا رسیدن به همگرایی برای سطح 9 برابر 95 است، درحالی که این عدد برای سطح 6 برابر 44 است.

## خروجی ها :

در هر سطح با هر دو روش بیان شده، بهترین خروجی در نسل ها انتخاب شده و به همراه امتیازهایشان در فایل خروجی چاپ می شود.

برای مثال خروجی مربوط به سطح 10 را در زیر می بینیم:

```
Method 1 output:  1000000000220000100100000002000010000000001010000020200010000200002100010001002000001000002000010001  187.76
Method 2 output:  1000000000220000100100000002000010000000001010000020200010000200002010010001002000001000002000100001  184.62
```

- اختلاف امتیاز ها در این نمونه به دلیل نحوه متفاوت محاسبه امتیاز ها در دو روش است ولی این امتیاز ها در هر روش بیشترین امتیازی هستند که می توان کسب کرد.
- علاوه بر این می بینیم که بعضی از حرکات در دو روش متفاوت هستند. این تفاوت باعث برتری هیچ یک بر دیگری نمی شود زیرا کروموزوم ها در هر دو مکان امتیازی را کسب می کند که برابر است ولی به دلیل اینکه می سوزند نمی توانند هر دو امتیاز را کسب کنند پس مجبورند یکی را انتخاب کنند که باعث برابری هر دو می شود.