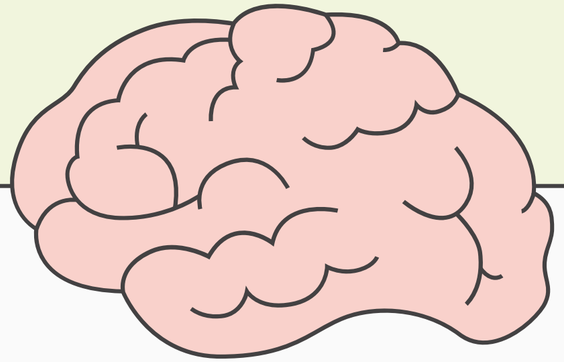
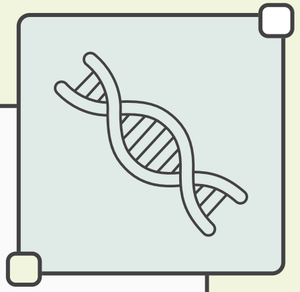
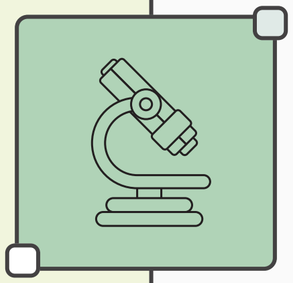




TSP Problem



Minseo Kang

2026.01.29

지난 GA 과제에서 보완점

1. 토너먼트 연산 구현
2. 알고리즘 종료조건(수렴) 제대로 구현하기

```python
    # r < t 이면 좋은 해 선택, 아니면 나쁜 해 선택
    def selection_operater(self, population):  1개의 사용 위치
        t = self.params['TOUR_T']

        def tourna():
            x1, x2 = random.sample(population, k: 2)  # 두 염색체 선택 (중복 없이)
            if x1[1] >= x2[1]:
                good, bad = x1, x2
            else:
                good, bad = x2, x1

            r = random.random()  # [0,1)
            if r < t:
                return good[0]  # chromosome
            else:
                return bad[0]   # chromosome

        mom_ch = tourna()
        dad_ch = tourna()
        return mom_ch, dad_ch
```

```python
# 최종 출력
best = population[0]
conv_ch, conv_cnt, conv_ratio = self.convergence_ratio(population)
print(
    "탐색이 완료되었습니다.\t 최종 세대수: {},\t 최종 해(상위1): {},\t 최종 적합도: {}\n"
    "수렴 해: {}, 수렴 개수: {}/{}, 수렴 비율: {:.3f}".format(
        *args: generation, best[0], best[1],
        conv_ch, conv_cnt, self.params["POP_SIZE"], conv_ratio
    )
)
```

```
population 평균 fitness: 1011.7
population 평균 fitness: 1012.3
population 평균 fitness: 1013.0
population 평균 fitness: 1014.4
population 평균 fitness: 1014.7
population 평균 fitness: 1015.0
탐색이 완료되었습니다.  최종 세대수: 16,  최종 해(상위1): 1111110111,  최종 적합도: 1015
수렴 해: 1111110111, 수렴 개수: 10/10, 수렴 비율: 1.000

종료 코드 0(으)로 완료된 프로세스
```

정답 해가 아니어도 일정 비율 이상 수렴시 종료됨

params 추가된 부분

```
# 파라미터 설정
params = {
    'MUT_RATE': 0.2,
    'POP_SIZE': 80,
    'NUM_OFFSPRING': 40,
    'TOURNAMENT_T': 0.6,
    'CONV_CHECK': 10,  # 몇 세대마다 수렴 검사할지(시간 제한이 있으니까)
    'CONV_MIN_GEN': 50,  # 너무 이른 조기종료 방지용 최소 세대
    'CONV_DOM_RATIO': 0.90,  # 90% 이상이면 수렴으로 판단
}
```

Nearest Neighbor(최근접 이웃) greedy 알고리즘

- greedy algorithm - 선택의 순간마다 당장 눈앞에 보이는 최적의 상황만을 쫓아 최종적인 해답에 도달
- TSP problem으로 치환 - “현재 도시에서 가장 가까운 미방문 도시”를 계속 선택

```
    # 1. 시작 도시 결정
    start = 0
    n = self.n_cities
    dm = self.dist_matrix

    visited = [False] * n
    tour = [start]
    visited[start] = True
    cur = start
```

GPT says...
기존에는 굳이 코드 한 줄 더 하는 것보다
매번 호출하는 방식을 선택했으나
time limit가 있는 상황에서는 매번 계속 호출하는 것보다
지역변수로 잡아야 접근 비용이 훨씬 낮다고 알려줌

Nearest Neighbor(최근접 이웃) greedy 알고리즘

- greedy algorithm - 선택의 순간마다 당장 눈앞에 보이는 최적의 상황만을 쫓아 최종적인 해답에 도달
- TSP problem으로 치환 - “현재 도시에서 가장 가까운 미방문 도시”를 계속 선택

```
# 2. 방문하지 않은 도시 중 가장 가까운 곳 선택 (반복)
for k in range(n - 1):
    nxt = None
    best_d = float("inf") # GPT : 아직 최소거리가 없으니 무한대로 두고 첫 후보가 무조건 갱신되게 함
    for j in range(n):
        if not visited[j]:
            d = dm[cur][j]
            if d < best_d:
                best_d = d
                nxt = j
    tour.append(nxt)
    visited[nxt] = True
    cur = nxt
```

```
best_len = self.get_fitness(tour) # 3. 마지막에 시작 도시로 복귀
print("Greedy Solution 탐색 완료")
print(f"  [Greedy start=0] len={best_len:.4f}, tour={tour}")
```

Nearest Neighbor(최근접 이웃) greedy 알고리즘

- greedy algorithm - 선택의 순간마다 당장 눈앞에 보이는 최적의 상황만을 쫓아 최종적인 해답에 도달
- TSP problem으로 치환 - “현재 도시에서 가장 가까운 미방문 도시”를 계속 선택

```python
def get_fitness(self, chromosome):  3개의 사용 위치
    # 경로의 총 길이를 계산하고, 적합도(역수 등)를 반환
    distance = 0
    for i in range(self.n_cities):
        a = chromosome[i]
        b = chromosome[(i + 1) % self.n_cities]  # 마지막 -> 시작 포함
        distance += self.dist_matrix[a][b]
    return distance
```

마지막 > 시작은 get_fitness에서 처리됨
알고리즘에서는 따로 복귀 X

selection_operater() : 토너먼트 연산 구현

```python
class TSP_GA:  1개의 사용 위치
    def selection_operater(self, population):  1개의 사용 위치
        #  if (t > r) x1 선택 else x2 선택

        t = self.params['TOURNAMENT_T']

        def tourna():
            # 1) 두 개체 무작위 선택 (독립추출)
            ch1, fit1 = population[self.rng.randrange(len(population))]
            ch2, fit2 = population[self.rng.randrange(len(population))]

            # 2) x1이 더 좋다고 가정 (거리 최소화)
            if fit2 < fit1:
                ch1, ch2 = ch2, ch1
                fit1, fit2 = fit2, fit1

            # 3) 확률적 선택
            r = self.rng.random()  # [0,1)
            if t > r:
                return ch1  # 더 좋은 해 선택
            else:
                return ch2  # 덜 좋은 해 선택

        mom_ch = tourna()
        dad_ch = tourna()
        return mom ch. dad ch
```

2) 후보 2개 뽑기 (토너먼트 참가자)

python 코드 복사

```python
ch1, fit1 = population[self.rng.randrange(len(population))]
ch2, fit2 = population[self.rng.randrange(len(population))]
```

- `randrange(len(population))` 로 **랜덤 인덱스 하나**를 뽑고,
- 그 인덱스의 개체를 가져와.
- 두 줄이니까 **두 개체를 무작위로 2명 뽑는 것**(토너먼트 참가자 2명).

10s 동안 생각함 ›

`rng` 는 **random number generator(난수 생성기)**의 줄임말이에요.
즉 `self.rng = random.Random(seed)` 는 "이 객체 전용 랜덤 뽑기기"를 하나 만들어 둔 겁니다.

왜 굳이 `self.rng` 를 쓰냐?

- **seed(시드)**를 고정하면, 매번 똑같은 난수 흐름이 나와서 **실험 재현**이 됩니다.
- `random.Random(seed)` 로 만든 `self.rng` 는 **전역 random과 분리**돼서, 다른 코드가 `random` 을 써도 영향을 덜 받아요.

마지막 > 시작은 get_fitness에서 처리됨
알고리즘에서는 따로 복귀 X

crossover_operater() : Order Crossover 구현

1) 순서교차(Order Crossover)

임의의 두 절단점을 지정하여 한 부모(P1)의 중간부분은 그대로 자손에 상속하고 다른 한 부모(P2)에서는 부모 P1의 중간 부분에 있는 인자를 제외하고 **상대적 순서를 보존**하여 한 자손 O1을 생성하는 교차

P1	1	2	3	4	5	6	7	8	9
P2	3	1	7	8	4	6	9	5	2

한 부모(P1)의 절단점 중간부분으로 자손 O1을 만든다.

O1	X	X	X	4	5	6	7	X	X

P2의 두번째 절단점부터 경로를 차례로 나열하고 O1에 상속한 인자 {4, 5, 6, 7}을 제거하면 {2, 3, 1, 8, 9}가 남는데, 이를 O1의 두번째 절단점 이후에 차례로 복사하면 자손 O1은 아래와 같이 생성이 된다.

O1	1	8	9	4	5	6	7	2	3

동일한 방법으로 자손 O2를 생성하면 아래와 같다.

```python
def crossover_operater(self, mom, dad):  1개의 사용 위치
    # Order Crossover : 순열을 깨지 않으면서 교차 연산
    n = self.n_cities

    l = self.rng.randrange(n)
    r = self.rng.randrange(n)
    while r == l:
        r = self.rng.randrange(n)
    if l > r:
        l, r = r, l

    child = [-1] * n

    # 1) mom의 구간 복사
    child[l:r + 1] = mom[l:r + 1]
    used = set(child[l:r + 1])

    # 2) dad 순서대로 나머지 채우기
    idx = (r + 1) % n # r+1부터 채우기 시작할 것임
    for city in dad:
        if city in used:
            continue # 중복 스킵
        while child[idx] != -1:
            idx = (idx + 1) % n
        child[idx] = city
        used.add(city)

    return child
```

idx에 따라서 탐색
mom 연속부분 + dad 남은 인자 집어넣기
= 자손 완성

mutation_operater() : inversion 방식

```
def mutation_operater(self, chromosome):  1개의 사용 위치
    n = self.n_cities
    a = self.rng.randrange(n)
    b = self.rng.randrange(n)
    if a > b:
        a, b = b, a
    if a == b:
        return chromosome[:] # 그대로

    ch = chromosome[:]
    ch[a:b+1] = reversed(ch[a:b+1]) # GPT,,
    return ch
```

mutation할 경우
특정 a, b 사이 구간을 reverse해서 변이를 줌
reversed 로 쉽게 뒤집기 가능!!

replacement_operater(), print_average_fitness() : 대치연산 구현

```python
def replacement_operator(self, population, offsprings):  1개의 사용 위치
    # population + offsprings 합친 뒤 거리 오름차순 정렬
    # 상위 POP_SIZE 다음 세대로 유지

    merged = population + offsprings
    merged.sort(key=lambda x: x[1])
    return merged[:self.params['POP_SIZE']]

def print_average_fitness(self, population):  1개의 사용 위치
    avg = sum(f for _, f in population) / len(population)
    best = population[0][1]
    print(f"[Gen] avg_len={avg:.4f}, best_len={best:.4f}")
```

merged[0]이 최선(best)
merged[-1]이 최악(worst)

좋은 해들만 남겨서 POP_SIZE 보존

Search() 함수 수렴을 위한 params 추가

- 1) time limit 를 고려한 수렴 체크 세대수 설정
- 2) ⭐GPT⭐ 선택압이 높아 발생하는 조기 수렴 방지를 위한 최소 세대 설정(최소 세대 이후부터 수렴 체크)
- 3) 수렴비율 설정(기본 0.9)

```python
# 파라미터 설정
params = {
    'MUT_RATE': 0.1,
    'POP_SIZE': 50,
    'NUM_OFFSPRING': 20,
    'TOURNAMENT_T': 0.75,
    'CONV_CHECK': 10,  # 몇 세대마다 수렴 검사할지(시간 제한이 있으니까)
    'CONV_MIN_GEN': 50,  # 너무 이른 조기종료 방지용 최소 세대
    'CONV_DOM_RATIO': 0.90,  # 90% 이상이면 수렴으로 판단
}
```

CONV_MIN_GEN = 설익은 수렴 방지

'CONV_MIN_GEN': 50 이거 빼면 안돼?

빼도 "코드는 돌아가고" 논리도 성립합니다. 다만 **빼면 조기 종료가 너무 빨리 걸릴 위험**이 있어서, 실험/발표 관점에서는 유지하는 게 보통 더 안전해요.

`CONV_MIN_GEN`의 역할

수렴 조건(최빈 투어 비율 ≥ 0.90)은 **진짜 수렴**이 아니라도 초반에 "우연히" 발생할 수 있습니다.

왜냐면 초반에는:

- 토너먼트 선택압이 있으면(`TOURNAMENT_T=0.75`) 좋은 개체가 빠르게 퍼지고,
- (μ+λ) 대치로 상위가 계속 살아남고,
- 우연히 같은 해가 중복 생성되면
- population 다양성이 짧은 시간에 급감할 수 있어요.

그런데 이 시점은 GA가 아직 **탐색할 여지**가 남아있을 가능성이 큽니다.
`CONV_MIN_GEN`은 이런 **"가짜 수렴(early convergence)"** 에서 너무 빨리 멈추는 걸 막는 안전장치예요.

수렴 종료조건 GA 설계

- tuple 변환을 통한 수렴 판단
- GPT에게 수렴 판단에 있어 부족한 부분 보완 요청 -> CONV_MIN_GEN 추가 후 if 조건문으로 디벨롭해줌

```
# 2-4) 수렴 종료조건
# population이 일정 비율 동일한 해가 되면 종료

if generation >= self.params.get("CONV_MIN_GEN", 0) and \
   generation % self.params.get("CONV_CHECK", 10) == 0: # GPT
    # 리스트를 tuple로 변환
    tours = [tuple(ch) for ch, fit in population]
    cnt = Counter(tours)
    rat = cnt.most_common(1)[0][1] / len(population)

    if rat >= self.params.get("CONV_DOM_RATIO", 1.0):
        # 수렴으로 판단 → 조기 종료
        print(f"[Converged] dominant_ratio={rat:.2f} at gen={generation}")
        break
```

실험용 기본 파라미터 설정

```
# 파라미터 설정
params = {
    'MUT_RATE': 0.2,
    'POP_SIZE': 80,
    'NUM_OFFSPRING': 80,
    'TOURNAMENT_T': 0.7,
    'CONV_CHECK': 10,  # 몇 세대마다 수렴 검사할지(시간 제한이
    'CONV_MIN_GEN': 50,  # 너무 이른 조기종료 방지용 최소 세대
    'CONV_DOM_RATIO': 0.9,  # 90% 이상이면 수렴으로 판단
    'LOG_INTERVAL_SEC': 0.2,
}
```

결과표: Greedy(start=0) vs GA, Cycle11, Cycle21

- 1) Cycle 11: std=0.0000 , 작은 공간에서는 GA가 5회 수행 모두 동일 해로 도달
- 2) Cycle 21: 개선율 약 26%
  도시 수가 늘어나자 Greedy 알고리즘보다 GA가 약 26% 더 짧은 경로 도달

```
==============================
[Experiment] cycle11.in
Greedy(start=0) = 362.6103
------------------------------
seed |  GA_best_len | improve(%)
---------------------------------
   0 |     345.8752 |       4.62
   1 |     345.8752 |       4.62
   2 |     345.8752 |       4.62
   3 |     345.8752 |       4.62
   4 |     345.8752 |       4.62
------------------------------
GA_best_len mean±std = 345.8752 ± 0.0000
improve(%)   mean±std = 4.62 ± 0.00
best_trace seed = 0 (best_len=345.8752)
==============================
```

```
==============================
[Experiment] cycle21.in
Greedy(start=0) = 552.8773
------------------------------
seed |  GA_best_len | improve(%)
---------------------------------
   0 |     399.7112 |      27.70
   1 |     406.1384 |      26.54
   2 |     406.1384 |      26.54
   3 |     406.1384 |      26.54
   4 |     406.1384 |      26.54
------------------------------
GA_best_len mean±std = 404.8530 ± 2.8744
improve(%)   mean±std = 26.77 ± 0.52
best_trace seed = 0 (best_len=399.7112)
==============================
```

결과표: Greedy(start=0) vs GA, Cycle51, Cycle101

- 1) Cycle 51 : 개선율 22.08%, 표준편차(Std): 13.86 , 시드별로 다른 local solution으로 수렴함
- 2) Cycle 101 : 개선율 9.59%, Greedy : 926.04 vs GA : 837.27

```
==============================
[Experiment] cycle51.in
Greedy(start=0) = 792.1301
------------------------------
seed |  GA_best_len | improve(%)
---------------------------------
   0 |     621.2543 |      21.57
   1 |     606.6362 |      23.42
   2 |     621.1435 |      21.59
   3 |     601.0036 |      24.13
   4 |     636.2549 |      19.68
------------------------------
GA_best_len mean±std = 617.2585 ± 13.8652
improve(%)   mean±std = 22.08 ± 1.75
best_trace seed = 3 (best_len=601.0036)
==============================
```

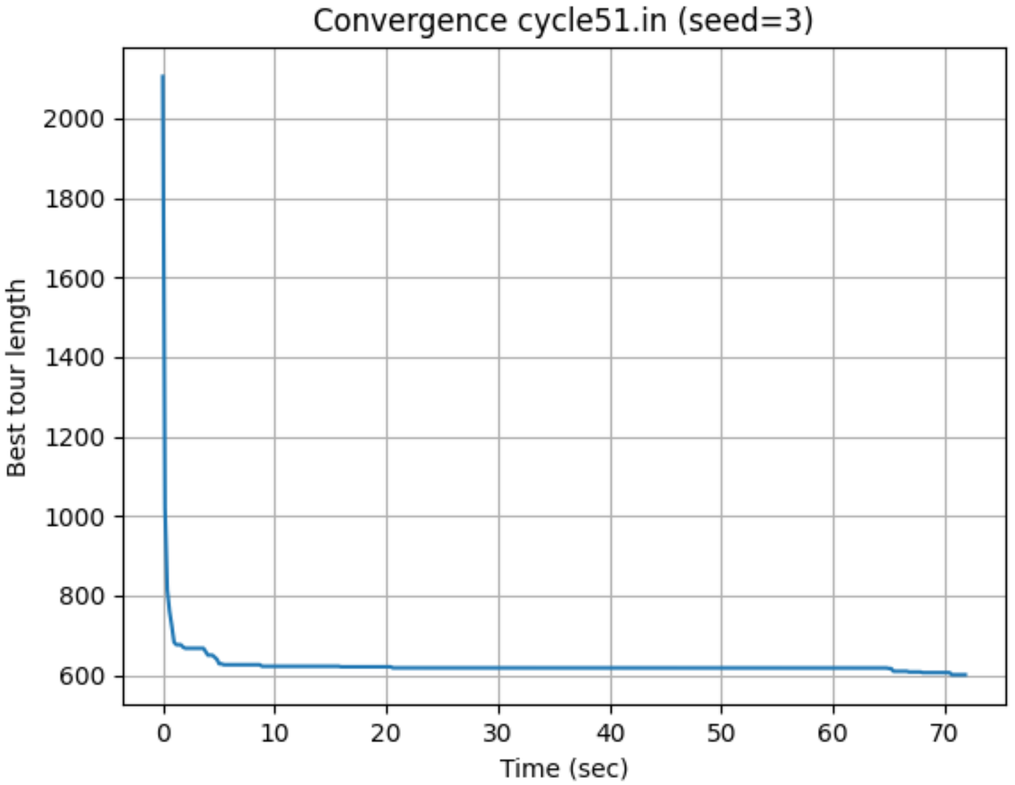
```
==============================
[Experiment] cycle101.in
Greedy(start=0) = 926.0428
------------------------------
seed |  GA_best_len | improve(%)
---------------------------------
   0 |     826.5866 |      10.74
   1 |     840.3236 |       9.26
   2 |     833.6240 |       9.98
   3 |     820.2811 |      11.42
   4 |     865.5422 |       6.53
------------------------------
GA_best_len mean±std = 837.2715 ± 17.4978
improve(%)   mean±std = 9.59 ± 1.89
best_trace seed = 3 (best_len=820.2811)
==============================
```

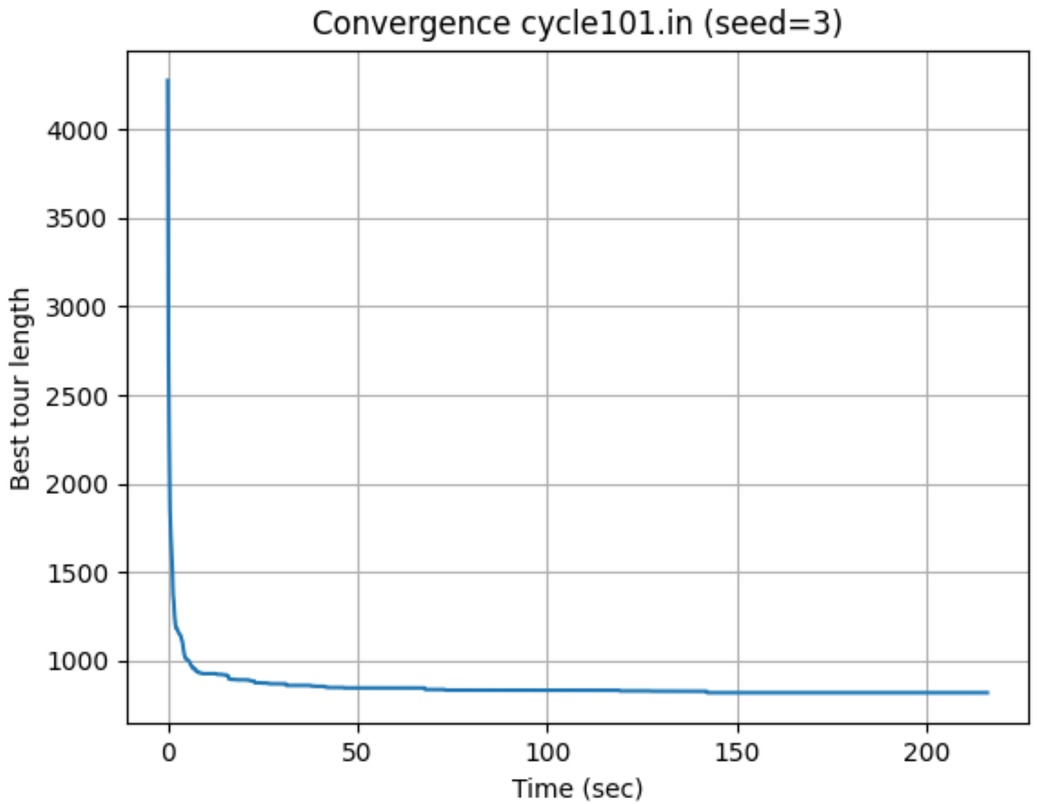
결과 요약 : Greedy vs GA

N=11, 21 : GA가 압도적으로 좋다! 특히 11에서는 매번 동일 해를 찾아냄
N=51 : Greedy로 찾지 못한 투어를 잘 찾아내면서 20% 이상 개선됨
N=101 : 다시 효율이 떨어지는 느낌 / 개선율 9% 대

수렴곡선 : cycle51, cycle101

- 1) 시작 직후에 급격하게 tour length가 짧아지지만 특정 시점 이후부터는 해의 품질이 더 이상 나아지지 않음. = > 설익은 수렴 의심됨
- 2) 두 사이클 모두 Elitism 으로 인해 해집단의 다양성을 빠르게 고갈한 것이 아닐지...
- 3) 후반부에서 탐색(Exploitation)이나 탈출(Exploration) 연산이 거의 안되는 것 같음





탐색 공간이 가장 넓은 Cycle51, Cycle 101로 파라미터 조정 실험 진행

MUT=0.2, TOUR=0.7, POP=80 (기존에 쓰던 설정)

실험 1 (MUT_RATE): 0.2 / 0.4
실험 2 (TOURNAMENT_T): 0.5 / 0.7
실험 3 (POP_SIZE(NUM_OFFSPRING)): 80 / 120 / 200

3. 실험 설계: 파라미터를 어떻게 바꿀까?

구조(Replacement 로직)를 바꾸는 건 대공사일 수 있으니, **파라미터 조절로 "다양성(Diversity)"을 강제 주입**하는 실험을 제안합니다.

[비교 실험 컨셉: Fast Convergence vs. High Diversity]

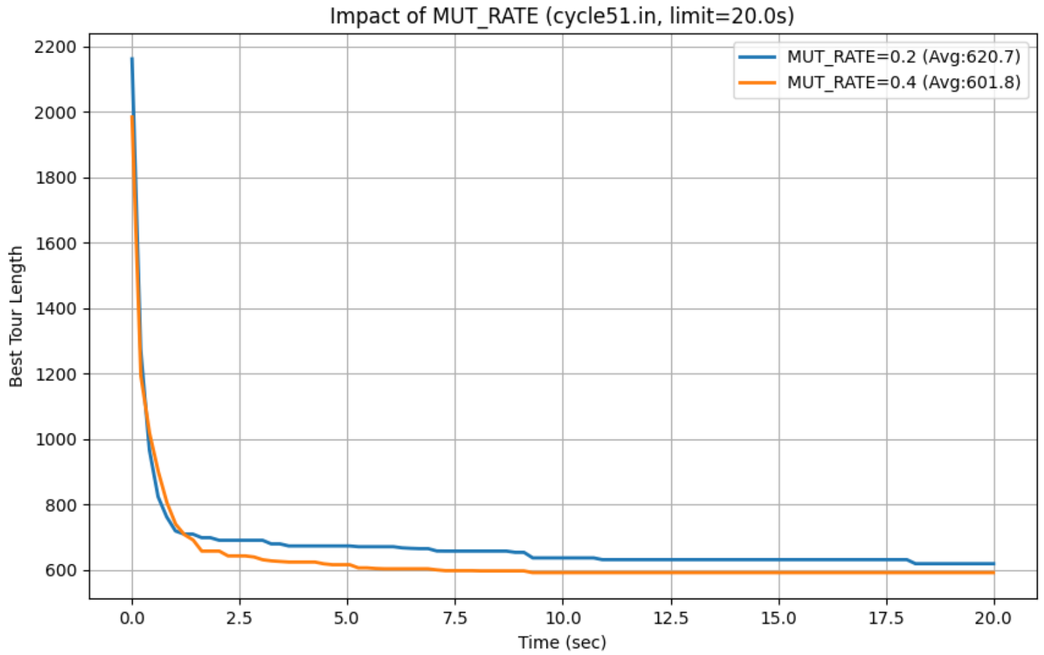
- **Group A (기존 - 조기수렴형)**: `POP=80`, `TOUR=0.7`, `MUT=0.2`
- **Group B (개선 - 다양성 중시형)**:
  - `POP_SIZE` : **150 ~ 200** (대폭 증가) → 초기 유전자 풀을 넓혀서 쉽게 획일화되지 않게 함.
  - `TOURNAMENT_T` : **0.6** (감소) → 1등이 아니어도 선택될 확률을 높여 "약간 부족하지만 독특한 경로"를 보존.
  - `MUT_RATE` : **0.3 ~ 0.4** (증가) → 고인 물이 되지 않도록 계속 뒤섞어줌.

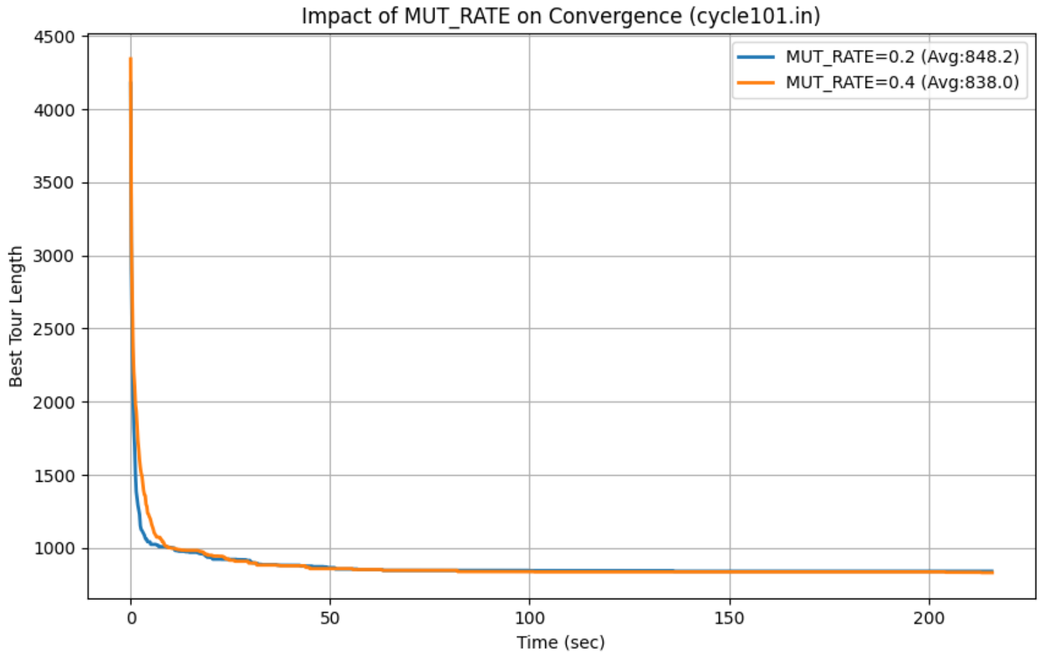
예상 시나리오: Group B는 초반 1~2초 동안은 A보다 성능이 안 좋아 보일 수 있습니다(그래프가 천천히 내려감). 하지만 시간이 지날수록 A는 멈춰있는데 B는 계속 내려가서, 결국 **최종 해는 B가 더 좋을 것**입니다.

GEMINI says,,,
다양한 파라미터가 있으니 그룹 A와 B로 나누어 비교하라고 했으나
단일 파라미터별로 생기는 변화를 확인하는 게 더 좋을 것 같아서 단일 파라미터별로 변경시키면서 실험하기로 결정함

MUT_RATE : 0.2 / 0.4

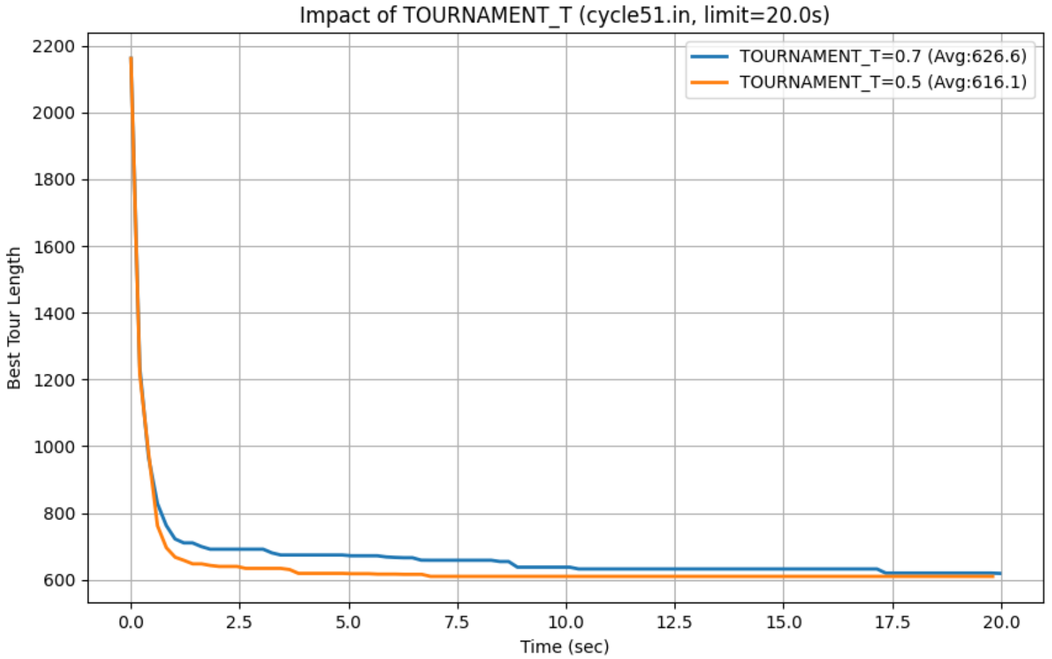
- 1) Cycle 51 : 기존에는 약 2.5초 시점에서 수렴하여 정체되었으나, 변이율이 높아지면서 지속적인 탐색(Exploration)이 발생,
  최종 해가 약 3% 개선됨 (620.7 → 601.8)
- 2) Cycle 101 : Cycle 51보다는 덜하지만, 유사한 경향

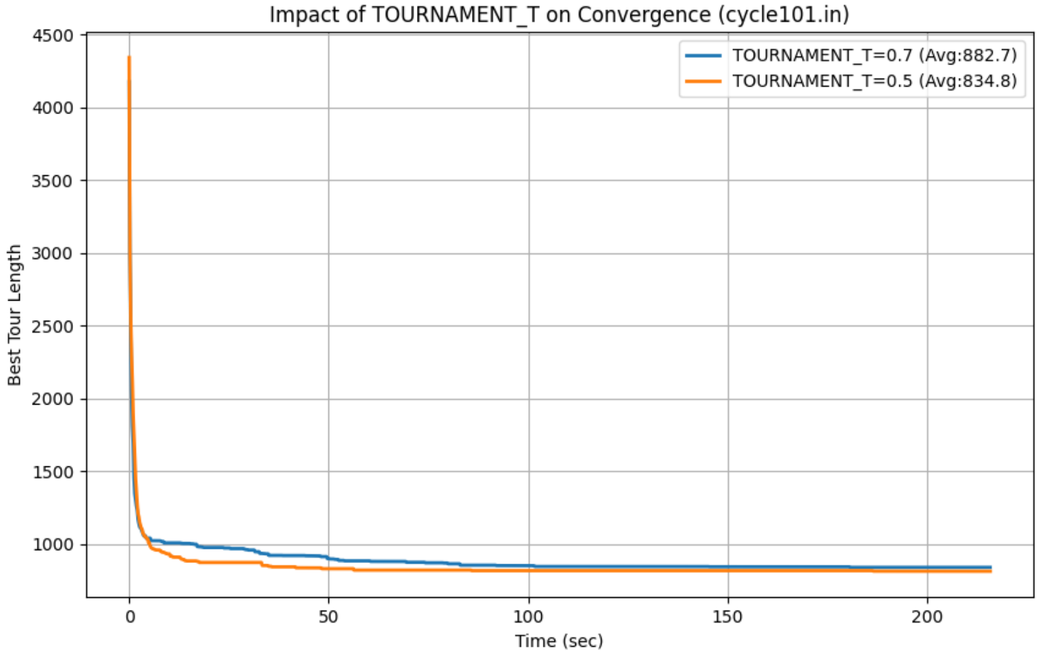




TOURNAMENT_T : 0.5 / 0.7

- 1) Cycle 51 : 선택압이 높은 0.7(파란선)은 초반 속도는 빠르나 조기 수렴함.
  반면, 0.5(주황선)는 초반 기울기는 완만해도 다양성을 유지하며 꾸준히 하락
- 2) Cycle 101 : 0.7 대비 0.5에서 압도적으로 우수한 최종 성능(882.7 vs 834.8) -> 약한 선택압이 장기적으로 더 유리함





POP_SIZE(=NUM_OFFSPRING) : 80 / 120 / 200

- 1) Cycle 51 : 80(파란선)은 유전자 풀이 좁아서 조기 수렴했고,
  200(초록선)은 120(주황선)과 비슷하게 수렴함 -> 무조건 큰 세대수가 좋지만은 않음