영남대학교
Yeungnam University

[REPORT HW2]

TSP 문제 풀이 프로그램

■ 과 목 명 : 인공지능
■ 담당교수 : 이기동 교수님
■ 제 출 일 : 2022-10-23
■ 학 과 : 로봇공학과
■ 학 번 : 22012383
■ 성 명 : 이윤서

1) Blind Search-BFS

BFS 알고리즘을 사용하여 탐색했다. 만약 맹목적 탐색을 위해 모든 경로의 경우의 수를 다 고려한다면 20!=2432902008176640000개의 경우의 수가 나오게 된다. 연산 시간과 메모리가 무한하다면 모든 경우의 수를 다 고려하여 가장 적합한 path를 구할 수 있지만, 현실 세계에서는 불가능한 가정이기 때문에 가지치기를 하는 형태로 코드를 구현했다.

select 함수로 다음으로 갈 수 있는 도시 후보들의 좌표와 현재 내 좌표를 비교한다. 그 후 거리가 가까운 순으로 정리해주고, 첫 번째 인덱스에 있는 도시로 가게 된다. 이런 형태의 가지치기를 한 이유는 도시찾기 알고리즘이라면 거리가 가장 가까운 도시로 가는 것이 최단 경로를 탐색하는 데에 적합하다고 생각했기 때문이다. 그리고 이미 갔던 도시는 다시 가지 않도록 하기위해 city 리스트에서 제거해주었다.

```
city = [] # 도시들을 저장해줄 list
level_max = 20 #깊이 level
start = []
path = []

def rand_city(city_n): # random으로 20개의 도시를 생성해주는 함수
  global city
  count = 0
  while count<city_n:
    x = random.randint(0,100)
    y = random.randint(0,100)
    new_city = [x,y]
    if new_city in city: #좌표가 중복되면 pass
      pass
    else:
      city.append(new_city) #새로운 좌표라면 추가 및 count+=1
      count = count+1
  print("city = ",city) # 도시 20개가 생성되면 출력

def select(now,go): #now는 나의 현위치, go는 남은 city
  distance = [] # 다음 도시 후보들과의 거리를 저장
  for i in range(len(go)):
    x = abs(now[0]-go[i][0])**2 # ax - bx
    y = abs(now[1]-go[i][1])**2 # ay - by
    distance.append(math.sqrt(x+y));
  distance_sort = sorted(distance) #거리 값이 가가운 순서대로 정열
  if len(distance_sort) == 1:
    return distance.index(distance_sort[0]) # 남은 도시 수가 하나라면 즉 탐색할 도시가 하나 남았다면 그냥 추가
  else:
    return distance.index(distance_sort[1])
    # 남은 city 중 현위치에서 가장 가까운 도시를 알려준다 [0]은 now도시, 따라서 1번 인덱스에 있는 도시를 return
```

```
####################main_BFS####################
rand_city(20) # 도시를 random으로 생성한다 100*100 좌표
start = copy.deepcopy(city[0]) # 출발지 도시를 저장
destination = copy.deepcopy(city[0]) # 도착지 도시를 저장

now = start #내가 현재 있는 도시
go = [] #내가 앞으로 갈 도시
now_level = 0 #path에 start(출발지) 도시만 있는 상황

while now_level<level_max: #깊이층이 20개가되면 종료
  index_n = select(now,city)
  next = city[index_n]

  path.append(now) #내가 왔던 곳은 경로에 추가
  city.remove(now) #내가 왔던 곳은 삭제
  now = next

  now_level += 1
path = path+[destination] #path의 마지막 지점은 도착지와 같기 때문에 위에서 저장해둔 리스트를 추가한다
print(path)
print(len(path))
```

코드의 실행결과를 확인하기 위해 turtle을 이용해 그림을 그려서 확인하였다.

출력1

city : [[27, 51], [80, 100], [3, 87], [6, 93], [53, 86], [13, 89], [41, 78], [29, 31], [69, 89], [58, 44], [79, 94], [80, 13], [47, 25], [94, 0], [64, 28], [62, 58], [47, 19], [39, 52], [2, 98], [34, 92]]

path : [[27, 51], [39, 52], [58, 44], [62, 58], [41, 78], [53, 86], [69, 89], [79, 94], [80, 100], [34, 92], [13, 89], [6, 93], [2, 98], [3, 87], [29, 31], [47, 25], [47, 19], [64, 28], [80, 13], [94, 0], [27, 51]]

time : 0.002698183059692383초

출력2

city : [[49, 21], [64, 73], [11, 21], [83, 35], [75, 58], [75, 71], [97, 60], [63, 8], [50, 99], [13, 77], [75, 36], [82, 81], [44, 3], [67, 69], [86, 90], [1, 11], [81, 54], [6, 45], [34, 94], [59, 12]]

path : [[49, 21], [59, 12], [63, 8], [44, 3], [11, 21], [1, 11], [6, 45], [13, 77], [34, 94], [50, 99], [64, 73], [67, 69], [75, 71], [82, 81], [86, 90], [97, 60], [81, 54], [75, 58], [75, 36], [83, 35], [49, 21]]

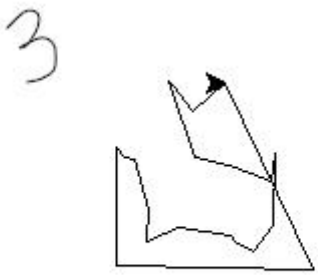
time : 0.0011839866638183594

출력3

city : [[55, 93], [78, 44], [40, 56], [99, 0], [80, 58], [79, 22], [1, 2], [27, 94], [69, 9], [30, 91], [16, 14], [61, 13], [17, 30], [1, 61], [58, 17], [10, 55], [39, 79], [60, 51], [32, 21], [5, 56]]

path : [[55, 93], [39, 79], [30, 91], [27, 94], [40, 56], [60, 51], [78, 44], [80, 58], [79, 22], [69, 9], [61, 13], [58, 17], [32, 21], [16, 14], [17, 30], [10, 55], [5, 56], [1, 61], [1, 2], [99, 0], [55, 93]]

time : 0.001621246337890625



출력4

city : [[98, 33], [92, 92], [60, 36], [38, 35], [27, 0], [17, 88], [88, 47], [60, 90], [49, 93], [83, 55], [95, 89], [61, 53], [77, 4], [5, 15], [84, 21], [2, 31], [32, 92], [64, 65], [73, 41], [55, 81]]

path : [[98, 33], [88, 47], [83, 55], [73, 41], [60, 36], [61, 53], [64, 65], [55, 81], [60, 90], [49, 93], [32, 92], [17, 88], [38, 35], [2, 31], [5, 15], [27, 0], [77, 4], [84, 21], [95, 89], [92, 92], [98, 33]]

time : 0.003914356231689453

2) 유전자알고리즘

```
class Chromosome:
  def __init__(self,g=[]):
    global start_city
    global city
    self.genes = g.copy()
    self.fitness = 0
    if self.genes.__len__()==0:
      self.path = random.sample(city[1:], len(city)-1)
      self.genes = [start_city] + self.path + [start_city]


  def cal_fitness(self): #적합도를 계산해주는 함수
    i = 0
    total_distance = 0
    distance = []
    for i in range(20):
      x = abs(self.genes[i][0] - self.genes[i+1][0])**2 #x
      y = abs(self.genes[i][1] - self.genes[i+1][1])**2 #y
      distance.append(math.sqrt(x+y));
    total_distance = sum(distance) #도착지까지 총 걸리는 거리를 비교
    self.fitness = 1/total_distance*10000
    #적합도는 높을수록 좋지만 거리값은 낮을수록 좋기때문에 역수를 취해줌
    return self.fitness
```

교재에 있는 selection 연산자는 룰렛으로 영역을 할당하여 화살표가 향하는 쪽의 염색체를 택하는 알고리즘이지만 가장 적합도가 뛰어난 함수의 특성을 계속 살려주는 것이 맞다고 판단하여 Elitist preserving selection으로 구현하였다. elitist preserving에 따라 현재 세대를 적합도가 높은 순으로 정렬하여 가장 적합도가 높은 유전자를 다음 세대에도 유지하는 형태이다. 또한 교재에 있는 것처럼 두 유전자의 일부를 섞는 알고리즘을 구현하니 갔던 도시를 또 가버리거나 도시가 누락되고, 중복없이 배열을 하자니 딱히 두 유전자의 특성을 살려주지 않는 것 같았다. 따라서 elitist preserving에서 선정된 가장 적합도가 높은 유전자 하나를 선택하여 부모로 선정하고 출발지와 도착지를 제외한 두 도시를 뽑아서 바꾸어주는 알고리즘을 구현했다.

```
# 유전자 crossover
def crossover(best1):
  parent = copy.deepcopy(best1)
  change_index1 = random.randint(2,19)
  change_index2 = random.randint(2,19)
  temp = 0

  temp = parent[change_index1]
  parent[change_index1] = parent[change_index2]
  parent[change_index2] = temp
  child = parent

  return (child)
```

돌연변이 연산자 또한 같은 방식으로 구현하였다.

```
def mutate(c):
  #print(c.genes)
  for i in range(city_n):
    if random.random() < mutation_rate:
      temp = 0
      change_index1 = random.randint(1,19) #a,b 도 포함 시작지와 도착지는 예외
      change_index2 = random.randint(1,19)
      temp = c.genes[change_index1]
      c.genes[change_index1] = c.genes[change_index2]
      c.genes[change_index2] = temp
```

한 세대의 구성은 다양성을 높이기 위해 세대 당 100개의 유전자를 가진다는 가정을 하였고, 1개는 elitist preserving에서 나온 유전자, 49개는 완전히 새로운 유전자, 50개는 부모 유전자에서 하나씩 섞은 유전자로 구성하였다.

```
while population[0].fitness<50:
  population.sort(key=lambda x: x.cal_fitness(), reverse=True)
# 부모의 수를 하나로 한다
  new_pop = []
  #print("old_적합도",population[0].fitness)
  b = population[0].genes
  #print("old_best=",b)
  #new_pop.append(population[0])
  for k in range(0,51):  #n개
    new = crossover(b) #new = list
    new_pop.append(Chromosome(new))
    #print(new_pop[k].fitness) == 0
  for j in range(0,24): #50-n개는 새로 생성
    new_pop.append(Chromosome())
  for c in new_pop: mutate(c)
  new_pop.append(Chromosome(population[0].genes))
```

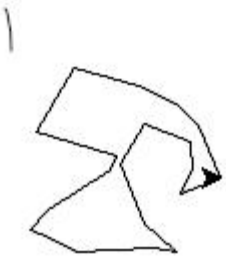
결과1

city = [[95, 39], [57, 66], [3, 62], [40, 43], [23, 2], [45, 46], [0, 13], [54, 85], [80, 57], [9, 23], [57, 16], [84, 65], [22, 94], [12, 8], [74, 75], [75, 31], [43, 50], [73, 2], [61, 4], [81, 43]]

path = [[95, 39], [84, 65], [74, 75], [54, 85], [22, 94], [3, 62], [43, 50], [40, 43], [9, 23], [0, 13], [12, 8], [23, 2], [61, 4], [73, 2], [57, 16], [45, 46], [57, 66], [80, 57], [81, 43], [75, 31], [95, 39]]

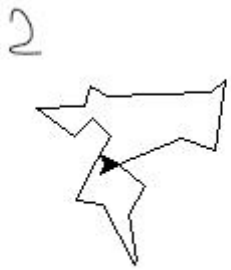
적합도 = 21.680330771301545

time = 22.21641707420349



결과2

city = [[43, 56], [29, 95], [39, 70], [74, 69], [22, 38], [2, 84], [96, 98], [51, 6], [48, 31], [91, 63], [89, 92], [26, 85], [35, 36], [32, 58], [21, 70], [37, 90], [52, 8], [52, 38], [56, 45], [30, 79]]
path = [[43, 56], [74, 69], [91, 63], [96, 98], [89, 92], [37, 90], [29, 95], [26, 85], [2, 84], [21, 70], [30, 79], [39, 70], [32, 58], [22, 38], [35, 36], [51, 6], [52, 8], [48, 31], [52, 38], [56, 45], [43, 56]]
적합도 = 26.085846957682474
time = 19.52083992958069



결과3

city = [[71, 65], [36, 47], [9, 55], [61, 81], [5, 59], [23, 72], [100, 60], [84, 92], [84, 72], [64, 10], [81, 47], [72, 6], [36, 63], [93, 45], [26, 32], [26, 65], [26, 51], [83, 2], [88, 56], [7, 89]]
path = [[71, 65], [61, 81], [23, 72], [26, 65], [9, 55], [5, 59], [7, 89], [84, 92], [84, 72], [100, 60], [88, 56], [81, 47], [93, 45], [83, 2], [72, 6], [64, 10], [26, 32], [36, 47], [26, 51], [36, 63], [71, 65]]
적합도 = 21.624546036948747
time = 19.513493299484253

결과4

city = [[64, 27], [88, 27], [56, 94], [82, 48], [11, 62], [47, 40], [48, 42], [46, 62], [43, 38], [78, 43], [22, 98], [49, 93], [84, 49], [31, 63], [68, 8], [72, 57], [93, 49], [54, 44], [20, 66], [31, 47]]
path = [[64, 27], [78, 43], [54, 44], [48, 42], [47, 40], [43, 38], [31, 47], [46, 62], [31, 63], [20, 66], [11, 62], [22, 98], [49, 93], [56, 94], [72, 57], [82, 48], [84, 49], [93, 49], [88, 27], [68, 8],

[64, 27]]
적합도 = 29.625337147430137
time = 20.88176918029785

결과 분석

BFS 알고리즘은 가까운 곳만 탐색하기 위해 가지치기를 하여 정확도가 낮게 나오지만 탐색 시간이 빨랐다. 유전자 알고리즘의 경우 세대 수와 개체수가 많을 수록 결과가 잘나오는 것으로 보였고, 결과가 잘나오는 만큼 출력시간이 BFS보다 오래 걸렸다. 따라서 BFS 알고리즘의 가지치기 방법을 좀 더 보완하였으면 탐색 양은 많아지지만 유전자 알고리즘과 비슷한 결과를 낼 수 있었을 것같다.