Caden Kroonenberg

Problem Set 4

EECS 649

2-12-22

**4.1**

a. Hill-climbing search

c. First-choice hill climbing

e. A random walk that moves to a successor state without concern for the value

**4.2**

Genotype refers to what genes an organism might have (e.g. xX genes) while phenotype refers to how those genes are expressed physically. For example, an organism might have a genotype of bB for iris color which results in a phenotype of blue irises.

For genetic algorithms, the specific values of weights and biases could be referred to as the genotype because these values may vary from organism to organism and may change via mutation. Furthermore, they are an internal value (like genes). The performance resulting from those weights and biases (genotype) can be referred to as a phenotype because it’s the outward expression of the genotype.

**4.3**

a. A fitness function for an elevator would minimize wait time for both people who are on the elevator on their way to their destination floor and who are waiting for the elevator to come to them. People who have been waiting for longer amounts of time should have some priority over those who haven’t been waiting as long.

b. A fitness function for use in evolving agents that control stop lights on a city main street would minimize wait time for cars at red lights, proportional to the number of cars waiting. For example, at the Massachusetts Street and 11th Street intersection, the wait time for drivers at a red light on Mass is lower than the wait time for drivers at a red light on 11th because Mass is a busier street. It is also important that 11th Street doesn’t have to wait too long at a red light, so a maximum waiting time should be implemented as well.

**4.4**

X is the set of classes: {Cn | 1 n N} where N is the # of classes.

D is a set of professor-room-time tuples: { (Pa, Rb, Tc) | 1 a A, 1 b B, 1 c C } where A is the # of professors, B is the # of rooms, and C is the # of possible time slots for classes.

C: For any two classes, Cx and Cy, who have values (Px, Rx, Tx) and (Py, Ry, Ty).

If (Px = Py OR Rx = Ry) then Tx Ty.

**4.5**

The first action I made was to find the MRV which was F because it only had 2 possible values, 1 or 0. I used F = 1 because it was the least constraining value (it ruled out the fewest values in the remaining variables).

With F = 1, C3 = 1 and the next MRV, T, could be 5, 6, 7, 8, or 9 for C2 + 2T = O + 10 to hold. The possible O values for all T are shown below. I used these values to determine a T value.

The least constraining values were T = 7 and T = 8.

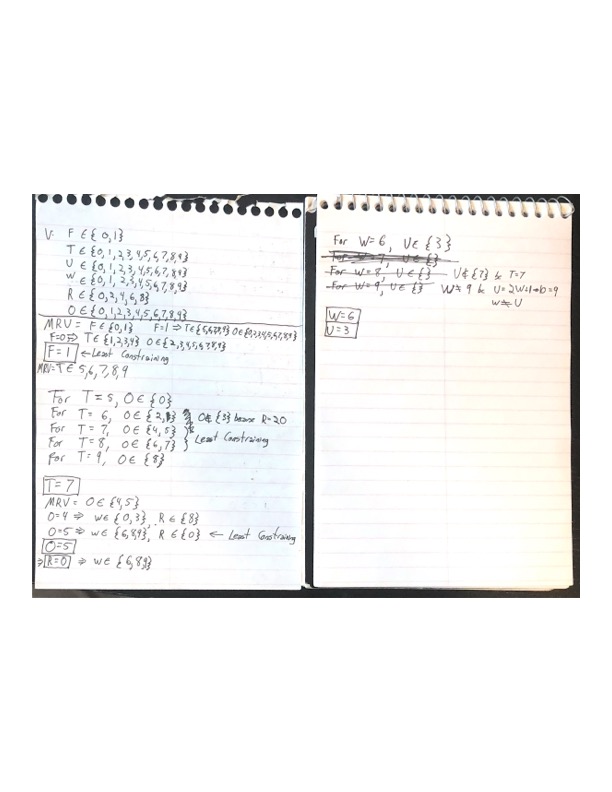
For T = 7, I found the MRV to be O. O must be 4 or 5 for C2 + 14 = O + 10 to hold.

The least constraining value for O was 5 (based on W values for each O value).

It immediately followed that R = 0 and C1 = 1 because 10\*C1 + R = O + O.

For R = 0 and O = 5, W must be 6 for 1 + 2W = U + 10 because the other possible W values (8,9) result in U values that violate the *Alldiff*(F, T, U, W, R, O) constraint:

If W = 8, U = 7. U cannot be 7 because T = 7, so W cannot be 8.

If W = 9, U = 9. U cannot be 9 because W = 9, so W cannot be 9.

Final Result: F = 1, T = 7, O = 5, R = 0, W = 6, U = 3

**4.6**

from math import sin, pow

import random

*def* F(*x*):

return 4 + 2\**x* + 2\*sin(20\**x*) - 4\*pow(*x*,2)

*def* roulette\_selection(*population*):

while True: # continue until return

max = sum([F(x) for x in *population*])

pick = random.uniform(0, max)

current = 0

# Return proportional to fitness

for x in *population*:

current += F(x)

if current > pick:

return x

population = [i\*0.01 for i in range(1,100)] # Initial population

N = 10

best = 0

for i in range(50): # For 50 generations

# Roulette Selection

select = []

while len(select) < N: # Ensure at least n individuals in population

select.append(roulette\_selection(population))

population = select.copy()

# Crossover

cross = []

for i in range(N):

# pick two parents

x = roulette\_selection(population)

y = roulette\_selection(population)

while x == y: # select unique parents

y = roulette\_selection(population)

a = random.random()

cross.append(a\*x+(1-a)\*y)

population = cross.copy()

# Mutation

mut = []

epsilon = 0.01

for x in population:

r = random.random()

if r <= 0.3: # x-epsilon w/ probability 0.3

if x-epsilon >= 0: mut.append(x-epsilon)

else: mut.append(0) # Clip to remain in [0,1]

elif 0.3 < r <= 0.7: # x w/ probability 0.4

mut.append(x) # copy w/ probability 0.4

else: # x+epsilon w/ probability 0.3

if x+epsilon <= 1: mut.append(x+epsilon)

else: mut.append(1) # Clip to remain in [0,1]

population = mut.copy()

for x in population:

if F(x) > F(best):

best = x

print('Best x = {}; F({}) = {}'.format(best,best,F(best)))

\*\*\* For my implementation without crossover, I just didn’t include the chunk of code starting with “# Crossover” and ending with “population = cross.copy()”

I used N = 10 and found a maximum F(x) at x = ~0.39 and F(0.39) = ~6.169.

The implementation without crossover was more inconsistent and wouldn’t always reach the same maximum as my crossover implementation; sometimes stopping around F(0.8) = 6.13.

The implementation with crossover would consistently find a maximum at x = ~0.39 with F(0.39) = ~6.169.

For each implementation I ran the algorithm for 50 generations, which is where convergence was typically found.

**4.7**

a. Random restart hill climbing:

import numpy as np

import random

*def* fitness(*queens*, *N*):

c = 0 # Number of conflicts

# For each queen

for q in range(len(*queens*)):

for q\_y in range(len(*queens*)):

q\_x = *queens*[q\_y]

if q\_y != q: # Check all other queens

if q\_x == *queens*[q]: # If queens are in same column

c += 1

if abs(q - q\_y) == abs(*queens*[q] - q\_x): # If queens share a diagonal

c += 1

return (*N*\*(*N*-1))/2 - c

*def* print\_grid(*queens*, *N*):

grid = np.full(*shape*=(*N*, *N*), *fill\_value*='-')

q\_x = 0

for q\_y in *queens*:

grid[q\_x][q\_y] = '\*'

q\_x += 1

for row in np.flip(grid, 0):

for cell in row:

print(cell, *end*=' ')

print()

*def* best\_successor(*queens*, *N*):

successors = []

for i in range(*N*): # For each queen

for j in range(*N*):

qc = *queens*.copy()

if qc[i] != j:

qc[i] = j

successors.append(qc)

best\_f = -1

for s in successors:

f = fitness(s,*N*)

if f > best\_f:

best\_s = s

best\_f = f

if best\_f == -1:

return *queens*

return best\_s

N = 8 # Grid size, number of queens

i = 0 # Epoch count

total\_i = 0 # Total epochs

success = False

while success == False:

# Random initial state

queens = []

for n in range(N):

queens.append(random.randint(0,N-1))

i = 0

while True and i < 50:

i+=1

total\_i+=1

next = best\_successor(queens,N)

# Plateau / no improvement

if fitness(next,N) <= fitness(queens,N):

break

queens = next.copy()

if fitness(queens,N) == (N\*(N-1))/2:

print(queens)

print('{} epochs'.format(i))

success = True

break

Simulated Annealing:

from math import exp, inf

import numpy as np

import random

*def* fitness(*queens*, *N*): # Same as random restart hill climbing

*def* schedule(*t*): # Test schedule function

# calculate temperature for current epoch

return 0.75\*(pow(*t*,-0.5)-0.025)

*def* rand\_successor(*queens*, *N*): # Return a set of queens with a random queen moved

q\_copy = *queens*.copy() # copy of queen positions

q = random.randint(0,*N*-1) # random queen, q

q\_y = q\_copy[q] # q position

while q\_y == q\_copy[q]: # choose new position for q

q\_y = random.randint(0,*N*-1)

q\_copy[q] = q\_y

return q\_copy

N = 8 # Grid size, number of queens

queens = [] # list of queen x-coordinates; queen[y] = x

for n in range(N): # Random start states for all N queens

queens.append(random.randint(0,N-1))

T = inf # Temperature

t = 1 # Epoch

while True:

T = schedule(t)

if T == 0: # Stop annealing when Temperature reaches 0

print('T == 0. No more annealing to be done')

print('There are {} conflicts.'.format((N\*(N-1)/2) - fitness(queens,N)))

print(queens)

break

next = rand\_successor(queens, N) # Pick a random successor

delta = fitness(next,N) - fitness(queens,N) # Calculate change in fitness

r = random.random() # Random number 0-1

p = exp(delta/T) # Probability of accepting next set of queens

if delta > 0: # If next has higher fitness than current, update queens

queens = next.copy()

elif r <= p: # Set next to queens anyway w/ probability e^(delta/T)

queens = next.copy()

if fitness(queens,N) == N\*(N-1)/2: # Break loop if solution has been found

print(queens)

print('SUCCESS in {} iterations.'.format(t))

break

t += 1 # Iterate epoch

|  |  |  |
| --- | --- | --- |
| Method | Average Number of Epochs | Average Number of Evaluations |
| Random Restart Hill Climbing | 28 | 1580 |
| Simulated Annealing  schedule = 0.998\*Tt-1  T0 = 30 | 2173 | 6519 |
| Simulated Annealing  schedule = 0.75(t-0.5 - 0.025)  t := epoch | 593 | 1780 |
| Simulated Annealing  schedule = 0.75(Tt-1-0.5 - 0.025)  T0 = 100 | 867 | 2600 |

For this problem, I interpreted the successors of a state to be the states in which a single queen has moved from its previous location (to anywhere else in it’s row).