알고리즘과 자료구조 워크북

교과목명 : 알고리즘과 자료구조
차시명: 7차시 몬테카를로 시뮬레이션

◈ 담당교수: 신일훈 **(서울과학기술대학교)**

◉ 세부목차

학습에 앞서

■ 학습개요

몬테카를로 시뮬레이션은 최적의 해를 구하기 위해 임의성(randomness)를 활용한다. 즉 문제를 풀기 위한 시뮬레이션을 임의성에 기반하여 설계하고 이를 반복하여 수행함으로써 최적의 해를 도출한 방법이다. 본강에서는 몬테카를로 시뮬레이션의 개념을 이해하고 이를 주사위 확률 구하는 문제, 원주율을 계산하는 문제, 간단한 적분값을 계산하는 문제, 마지막으로 knapsack을 해결하는 문제에 적용한다.

■ 학습목표

1	몬테카를로 시뮬레이션의 개념을 이해한다.
2	몬테카를로 시뮬레이션을 활용하여 다양한 주사위 확률을 구할 수 있다.
3	몬테카를로 시뮬레이션을 활용하여 원주율, 적분값 등을 계산할 수 있다.
4	몬테카를로 시뮬레이션을 활용하여 knapsack 문제를 해결하는 알고리즘을 설계할 수 있다.

■ 주요용어

용어	해설
임의성 (randomness)	무작위성을 의미함. 가령 주사위를 굴리는 행위가 이에 해당한다.
난수	규칙없이 무작위로 발생시킨 수. 파이썬은 정수인 난수를 발생시키는 메서드, 실수인 난수를 발생시키는 메서드 등을 지원한다.
몬테카를로 시뮬레이션	최적해를 찾기 위한 방법 중 하나로서, 해를 찾는 시뮬레이션을 임의성에 기반하여 설계하고 시뮬레이션을 반복함으로써 문제를 해결하는 방법.
객체	파이썬과 같은 객체 지향 언어에서 클래스 타입의 변수를 의미한다.

학습하기

1. 몬테카를로 시뮬레이션 개념
   - randomness에 기반하여 시뮬레이션을 설계하고, 시뮬레이션을 여러 번 수행하여 특정 이벤트가 발생한 회수를 계수하여 해당 이벤트가 발생할 확률을 추정함
   - 장점: 충분한 시도를 통해, 수학적으로 계산이 어려운 문제도 해결 가능함
   - 예시
     - 포커 게임 확률
     - 생일이 같을 확률
     - 원주율 계산
     - …

2. 몬테카를로 시뮬레이션의 적용1: 주사위 확률

   가. 문제
   - 주사위를 굴렀을 때 1이 나올 확률은?

   나. 아이디어
   - 주사위를 굴리는 시뮬레이션을 구현한다. 이 때 시뮬레이션은 randomness에 기반해야 한다. 즉 실제 주사위처럼 특정 숫자가 나오는 패턴이 있으면 안 된다.
   - 해당 시뮬레이션을 충분히 많이 반복하고, 1이 나온 회수를 계수한다.
   - (1이 나온 회수) / (전체 시도 회수)를 1이 나올 확률로 추정한다.

   다. 알고리즘(의사코드로 표현)

```
def rollDice():
        num = randint(1, 6)       #generate an integer between 1 and 6
        return num

def calDiceProb(tries, target) :
        count = 0
        for i in range(tries):
                num = rollDice()
                if (num == target) :
                        count += 1
        return (count/tries)
```

   - rollDice()는 주사위를 한번 굴리는 행위를 모델링한다. 1~6사이의 임의의 정수를 반환한다.
   - calDiceProb() 함수는 주사위를 굴렸을 때 1이 나올 확률을 추정하기 위한 함수이다.
   - 인자로 전달된 tries는 시뮬레이션의 실행 회수를 전달한다.
   - 인자로 전달된 tries만큼 주사위를 굴리고 (rollDice() 호출), 1이 나올 때마다 count를 증가시킨다.
   - 따라서 count에는 1이 나온 횟수가 저장된다.
   - 1이 나올 확률은 count/tries가 된다.

   라. 알고리즘 최악 시간복잡도
   - calDiceProb() 함수는 인자로 전달된 tries만큼 for문을 반복한다. 따라서 시뮬레이션의 실행 회수를 N이라고 한다면 시간복잡도는 O(N)이다.

마. 파이썬 구현 및 실행

```
import random
def rollDice():
        num = random.randint(1, 6)
        return num

def calDiceProb(tries, target) :
        count = 0
        for i in range(tries):
                num = rollDice()
                if (num == target) :
                        count += 1
        return (count/tries)

print(calDiceProb(100, 1))
print(calDiceProb(1000, 1))
print(calDiceProb(10000, 1))
print(calDiceProb(100000, 1))
```

- random 라이브러리는 난수를 발생시키기 위함이다.
- randint(from, to) 함수는 from ~ to사이의 임의의 정수를 반환한다.
- 다음은 이 파이썬 코드를 스파이더에서 실행한 결과 화면이다.
- 출력 결과를 보면 100번, 1000번, 10000번, 100000번 실행했을 때의 확률이 각각 0.18, 0.151, 0.17, 0.16675로 1/6에 가까운 값이 출력됨을 알 수 있다.
- 정확도는 일반적으로 시뮬레이션의 반복 회수에 비례하여 증가하는 경향을 나타낸다.
- 임의의 난수를 발생시키므로 결과는 실행할 때마다 달라진다.

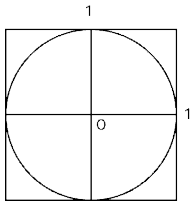
```
In [109]: runfile('D:/data/재정/주식소스/stock/result/
untitled3.py', wdir='D:/data/재정/주식소스/stock/result')
0.18
0.151
0.17
0.16675
```

3. 몬테카를로 시뮬레이션의 적용2: 원주율 계산

가. 문제

- 원주율(π)를 계산하시오.

나. 아이디어



- 한 변 2인 정사각형과 그 안에 내접한 반지름이 1인 원이 있다면,
- 정사각형 넓이 = 2 * 2 = 4, 원의 넓이 = π * r * r = π
- 즉, 해당 정사각형에 내접하는 반지름이 1인 원의 넓이가 원주율과 동일함을 알 수 있다.
- 그렇다면 반지름이 1인 원의 넓이를 어떻게 구할까?

- 정사각형의 넓이를 이미 알고 있으므로, 정사각형의 넓이와 원의 넓이의 비율을 알 수 있다면, 원의 넓이를 추정할 수 있다.
- 정사각형의 넓이와 원의 넓이의 비율을 어떻게 구할 것인가?
- 정사각형 안에 임의로 무수히 많은 점을 찍는다면, 어떤 점은 원 안에 있고, 어떤 점은 원 밖에 있을 것이다.
- 넓이의 비율은 임의로 찍은 점의 개수의 비율과 유사할 것이므로, 다음이 대체로 성립한다.
  - 정사각형 넓이 : 원의 넓이 = 정사각형 안의 점의 개수 : 원 안의 점의 개수
- 따라서 원의 넓이(π) = (원 안의 점의 개수 * 정사각형의 넓이) / 정사각형 안의 점의 개수
  = (원 안의 점의 개수 * 4) / 정사각형 안의 점의 개수
  = (부채꼴 안의 점의 개수 * 4) / 작은 정사각형 안의 점의 개수
- 다음은 무작위로 점을 찍는 행위를 시뮬레이션으로 어떻게 구현할 것인가이다.
- 점이 2차원 평면 상에 찍히게 되므로 0~1사이의 임의의 실수를 2개 발생시켜 각각 x, y 좌표로 사용하면 점을 찍는 행위가 된다.
- 점이 부채꼴 안에 있는지 여부는 원점에서 점까지의 거리가 1이하인지를 검사하면 된다. 1이하이면 원 안에, 즉 부채꼴 안에 있음을 의미한다.

다. 알고리즘(의사코드로 표현)

```
def calPI(tries):
        insideCnt = 0
        for i in range(tries):
                x = random(0, 1)      # generate a random number between 0~1
                y = random(0, 1)
                dist = sqrt(x*x + y*y)      # calculate the distance from the origin
                if (dist <= 1):
                        insideCnt += 1
        return (4*insideCnt/tries)
```

- 점을 찍는 행위를 모델링하기 위해, 0~1 사이의 임의의 실수를 2개 발생시킨다. (random())
- sqrt(x*x + y*y)는 원점에서 해당 점까지의 거리를 계산한다.
- 이 값이 1 이하이면 점이 부채꼴 안에 있음을 의미한다.
- 부채꼴 안에 있는 점의 개수는 insideCnt에 저장된다.
- 따라서 원주율은 (4*insideCnt/tries) 수식에 의해 계산된다.

라. 알고리즘 최악 시간복잡도

- calPI() 함수는 인자로 전달된 tries만큼 for문을 반복한다. 따라서 시뮬레이션의 실행 회수를 N이라고 한다면 시간복잡도는 O(N)이다.

마. 파이썬 구현 및 실행

```
import random
def calPI(tries = 10000):
        insideCnt = 0
        for i in range(tries):
                x = random.random()
                y = random.random()
                dist = (x*x + y*y)**0.5
                if (dist <= 1):
                        insideCnt += 1
        return (4*insideCnt/tries)

print(calPI(10))
print(calPI(100))
print(calPI(1000))
print(calPI(10000))
print(calPI(100000))
print(calPI(1000000))
```

- calPI() 함수는 알고리즘과 거의 유사하다.
- random() 함수는 0~1 사이의 실수를 반환한다.
- 다음은 이 파이썬 코드를 스파이더에서 실행한 결과 화면이다.
- 출력 결과를 보면 시뮬레이션의 반복 회수에 비례하여 실제 원주율에 가까운 값이 출력되는 경향을 나타낸다.
- 임의의 난수를 발생시키므로 결과는 실행할 때마다 달라진다.

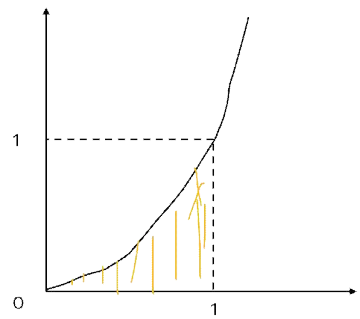
```
In [113]: runfile('D:/data/lecture/
파이썬으로배우는자료구조와알고리즘/code/알고리즘/untitled1.py',
wdir='D:/data/lecture/파이썬으로배우는자료구조와알고리즘/code/
알고리즘')
2.8
2.92
3.104
3.128
3.15008
3.14192
```

4. 몬테카를로 시뮬레이션의 적용3: 적분 계산

가. 문제

- y = x * x 을 0부터 1까지 적분한 값을 계산하시오.

나. 아이디어



- 빗금친 부분의 넓이가 구하고자 하는 적분값이다.
- 정사각형의 넓이 : 빗금 넓이 = 정사각형 안의 점의 개수 : 빗금 안의 점의 개수

- 정사각형의 넓이가 1이므로 다음이 성립한다.
  - 빗금 넓이 = 빗금 안의 점의 개수 / 정사각형 안의 점의 개수
- 따라서 무수히 많은 점을 찍어서 빗금 안의 점의 개수를 세면 적분값을 추정할 수 있다.
- 빗금 영역 안에 있는 지의 판단은 타겟 점의 x, y 좌표가 다음을 만족하는지를 체크하면 된다.
  - if (y <= x * x) => 빗금 영역 안에 있음.

다. 파이썬 구현 및 실행

```
import random
def callntegral(tries = 10000):
        insideCnt = 0
        for i in range(tries):
                x = random.random()
                y = random.random()
                if (y <= x * x):
                        insideCnt += 1
        return (insideCnt/tries)

print(callntegral(10))
print(callntegral(100))
print(callntegral(1000))
print(callntegral(10000))
print(callntegral(100000))
print(callntegral(1000000))
```

- callntegral() 함수는 인자로 전달된 tries만큼 for문을 반복한다.
- for 문 안에서는 점을 찍는 행위를 모델링하기 위해, 0~1 사이의 임의의 실수를 2개 발생시킨다. (random())
- y가 x*x 이하라면 빗금 영역 안에 있으므로 insideCnt를 증가시킨다.
- 적분값은 (insideCnt/tries) 수식에 의해 계산된다.
- 다음은 이 파이썬 코드를 스파이더에서 실행한 결과 화면이다.
- 출력 결과를 보면 시뮬레이션의 반복 회수에 비례하여 실제 적분값(1/3)에 가까운 값이 출력되는 경향을 나타낸다.
- 임의의 난수를 발생시키므로 결과는 실행할 때마다 달라진다.

```
0.3
0.26
0.321
0.3338
0.33658
0.333659
```

5. 몬테카를로 시뮬레이션의 적용4: 배낭(knapsack) 문제

가. 문제

- 배낭 문제 정의 및 기본적인 클래스 정의는 6차시에서 공부한 내용과 동일하므로 생략한다.

나. 아이디어

- 각 아이템의 포함 여부를 임의로 결정한다. (random 함수 활용)
- 단, 아이템을 포함했을 시, 최대 허용 무게를 초과하면 해당 아이템은 무조건 포함하지 않는다.

- 임의로 결정하므로 한 번만 시도해서는 최적값을 구할 수 없다.
- 따라서 이러한 시뮬레이션을 충분히 많이 수행하고 그 중에서 최적의 케이스를 최적값으로 간주한다.

다. 파이썬 구현 및 실행

- Item 클래스 및 Knapsack 클래스의 기본적인 정의는 6차시에서 공부한 것과 동일하므로 생략한다.
- 다음의 함수들을 기존의 Knapsack 클래스에 추가로 구현한다.

- findBestCaseMontecarlo() 함수 구현

```
def findBestCaseMontecarlo(self, tries=10000):
        max_value = 0
        best_chosen_items = []
        best_weight = 0

        for i in range(tries):
                (value, weight, chosen_items) = self.select_items()
                if (value > max_value) :
                        max_value = value
                        best_weight = weight
                        best_chosen_items = list(chosen_items)
        return (max_value, best_weight, best_chosen_items)
```

- findBestCaseMontecarlo() 함수는 시뮬레이션을 여러 번 수행하여 최적의 케이스를 찾기 위한 함수이다.
- max_value, best_chosen_items, best_weight는 모두 최적의 케이스를 저장하기 위한 변수이다.
- 인자로 전달된 tries만큼 for 문을 반복한다.
- for 문 안에서는 select_items() 함수를 호출하여 임의로 아이템들을 선택하고 선택된 아이템들과 그 때의 가치의 총합과 무게의 총합을 반환받는다.
- 만약 가치의 총합이 현재의 max_value보다 크다면 이 경우가 현재까지 최적이므로 해당 경우를 max_value, best_weight, best_chosen_items에 각각 저장한다.
- for 문 반복 후에는 최적의 케이스가 max_value, best_chosen_items, best_weight에 저장되어 있으므로 해당 값들을 반환한다.

- select_items() 함수 구현

```
def select_items(self):
        value = 0; weight = 0
        chosen_items = []

        for item in self.items:
                if (weight + item.weight <= self.max_weight) :
                        roll = random.randint(1, 100)
                        if (roll % 2 == 0):
                                chosen_items.append(item.name)
                                weight += item.weight
                                value += item.value
        return (value, weight, chosen_items)
```

- 모든 아이템에 대해 포함 여부를 난수를 통해 결정한다.
- 즉 난수가 짝수이면 아이템을 추가한다.

- 단 아이템을 포함할 시 총 무게가 허용 무게를 초과하면 해당 아이템은 무조건 선택하지 않는다.
- 선택된 아이템들과 가치의 총합, 무게의 총합을 반환한다.

- 다음은 작성한 몬테카를로 시뮬레이션을 실행하기 위한 코드이다. (100번 반복함)

```
names = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
values = [10, 30, 20, 14, 23, 11, 15, 18]
weights = [5, 8, 3, 7, 9, 2, 6, 1]
max_weight = 20
knapsack = Knapsack(names, values, weights, max_weight)
(value, weight, items) = knapsack.findBestCaseMontecarlo(100)
print(value, weight, items)
```

- 다음은 위 파이썬 코드를 스파이더에서 실행한 결과 화면이다.
- 배낭의 최대 무게가 20일 때, 최적의 조합은 B, C, F, G, H를 선택한 경우이며 그때의 가치의 총합은 94이다.

```
In [114]: runfile('D:/data/
lecture/
파이썬으로배우는자료구조와알고리즘/
code/알고리즘/untitled1.py',
wdir='D:/data/lecture/
파이썬으로배우는자료구조와알고리즘/
code/알고리즘')
94 20 ['B', 'C', 'F', 'G', 'H']
```

연습문제

1. 몬테카를로 시뮬레이션의 개념을 설명하시오.

정답 : 최적해를 찾기 위한 방법 중 하나로서, 해를 찾는 시뮬레이션을 임의성에 기반하여 설계하고 시뮬레이션을 반복함으로써 문제를 해결하는 방법.

해설 : 정답 참조

2. knapsack 문제를 해결할 때, brute-force 기법 대비 몬테카를로 시뮬레이션의 장점과 단점을 설명하시오.

정답 : 장점은 시뮬레이션의 반복 횟수를 brute-force보다 더 적게 조정할 수 있다는 점이다. 즉 더 빠른 시간 안에 수행을 마칠 수 있다. 단점은 임의성에 기반하고 모든 해 공간을 탐색하지 않으므로 최적해를 찾지 못할 수 있다는 점이다. 최적해의 근사값을 일반적으로 찾게 된다.

해설 : 정답 참조

3. 두 개의 주사위를 굴러서 두 개의 주사위의 숫자가 같을 확률을 계산하기 위한 방법을 몬테카를로 시뮬레이션을 활용하여 설명하시오..

정답 : 두 개의 주사위를 굴리는 시뮬레이션은 1~6 사이의 정수 난수 두 개를 발생시키는 형태로 구현한다. 만약 두 개의 난수가 같으면 두 개의 주사위 숫자가 같은 이벤트에 해당한다. 따라서 해당 시

뮬레이션을 N번 수행하고, 이 중 두 개의 난수가 같은 경우를 계수하여 이를 N으로 나누면 확률이 계산된다.

해설 : 정답 참조

정리하기

1. 몬테카를로 시뮬레이션은 최적해를 찾기 위한 방법 중 하나로서, 해를 찾는 시뮬레이션을 임의성에 기반하여 설계하고 시뮬레이션을 반복함으로써 문제를 해결하는 방법이다.
2. 주사위 관련 확률을 몬테카를로 시뮬레이션을 활용하여 계산할 수 있다.
3. 원주율, 적분값 등을 몬테카를로 시뮬레이션을 활용하여 계산할 수 있다.
4. Knapsack을 해결하는 알고리즘을 몬테카를로 시뮬레이션을 활용하여 설계할 수 있다.

참고자료

- 파이썬 알고리즘, 최영규, 2021, 생능출판

다음 차시 예고

- 자료구조 및 연결리스트에 대해 학습한다.