Travel Salesman Problem

Algorithm **analysis**

INDEX

1. TSP 알고리즘 분석

1.1 TSP 코드 리뷰

1.2 TSP BS와 DP 실행시간 분석 (n에 따른 그래프 및 빅 오 노테이션)

1.3 TSP BS bound 설정에 따른 성능 예상과 실제 구현

1.3.1 row

1.3.2 col

1.3.3 average

2. TSP 몬테카를로

1.1 몬테카를로의 정의

1.2 몬테카를로로 TSP 문제를 해결할 수 있는가 자체에 대한 문제

1.3 몬테카를로 분석

1.3.1 몬테카를로 Bound 자체에 몬테카를로를 넣는 경우

1.3.2 몬테카를로 Bound에 쓰레기 값을 이용하고, 노드를 임의로 enque 하는 경우

1.3.3 무엇이 더 실제와 비슷한가?

1. TSP 알고리즘 분석

이 보고서에서 우리는 첫번째로 Traveling Salesman Person에 대한 Best Search 및 Dynamic Programing을 적용한 알고리즘을 Python을 이용하여 구현하고, 알고리즘의 연산 속도에 대해 논할 것입니다.

추가로 탐색 할 내용은 bound를 계산 시 row, col, average를 이용하는 세가지 방법이다. 구동 방식에 대해 서술한 뒤, 세 bound 방법의 평균적인 실행시간을 비교해본다.

1. 코드

\*\* 부록 DP, BS 부분 참고

1. Best Search과 Dynamic Programming 실행시간 분석 & 비교

|  |  |  |  |
| --- | --- | --- | --- |
|  | n | BS | DP |
|  | 2 | 0.000226 | 0 |
|  | 3 | 0.000151 | 4.99E-05 |
|  | 4 | 0.000376 | 0.000125 |
|  | 5 | 0.000401 | 0.000802 |
|  | 6 | 0.001352 | 0.001657 |
|  | 7 | 0.003354 | 0.00623 |
|  | 8 | 0.009626 | 0.020843 |
|  | 9 | 0.012818 | 0.049823 |
|  | 10 | 0.042883 | 0.144843 |
|  | 11 | 0.108209 | 0.411776 |
|  | 12 | 0.285739 | 1.018201 |
|  | 13 | 0.440607 | 2.640385 |
|  | 14 | 0.736798 | 6.797047 |
|  | 15 | 1.703841 | 15.83577 |

실행시간 분석의 경우 input 값인 w matrix 에서 vertex n에 따라 실행시간을 분석하고, 시각화 및 Big-Oh function으로 표현한다.

실행시간 분석의 비교 코드엔 몬테카를로 시뮬레이션에 기초한, 각 vertex 간 edge를 랜덤으로 생성하여 20번씩 algorithm을 perform하는 방법을 사용한다.

우측 표는 BS와 DP의 n에 따른 실행시간 분석 데이터 입니다.

● Comparison between Dynamic Programming and

Dynamic Programming 의 경우 subset loop 속 n loop를 가지므로 전체 알고리즘은 Big-Oh () 를 따른다.

● Comparison between Best Search and

● Comparison between BS and DP

n이 증가할수록 Dynamic programming의 실행시간이 현저히 높다는 걸 알 수 있다.

Best Search Algorithm이 Dynamic Programming Algorithm보다 더 효율적입니다.

1. Bound 설정에 따른 성능 예상과 실제 구현

현재 구현한 TSP의 Best Search 알고리즘은 bound로 나가는 값(row)를 이용하였을 때 입니다. 이 외에도 bound로는 들어오는 값(col)과 나가는 값과 들어오는 값의 평균(average)을 이용할 수 있습니다. 알고리즘 수험 시간 때의 언급과 여러 자료의 내용을 종합하여 row, col, average 알고리즘을 각각 구현하고 그 값들을 구현해보았다.

예상:

들어오는 값(col)을 이용하였을 때, 결국에 각 노드에서의 bound값은 나가는 값(row)를 이용하였을 때와 차이가 없다. 엄밀하게 따져서 열을 기준으로 연산하게 되더라도, 결국에 이용하는 원소는 같기 때문이다. 또한, 들어오는 값과 나오는 값의 평균을 이용한다 하더라도, 결국에 어느 한 원소에서의 (들어오는 값 + 나가는 값) / 2의 값은, 그 앞의 원소의 나가는 값 / 2 부분과 그 뒤의 원소의 들어오는 값 / 2 부분으로 보충되기 때문에, 몇몇 산술적인 연산이 bound를 구할 때, 더 실행되는 것 이외에는 전체적인 bound와 노드 수에 큰 변화는 없을 것이다.

따라서 두 경우 모두 나가는 값을 이용한 연산에 비교하여 같은 노드, 큰 차이가 없는 실행 시간이어야 한다.

● Bound: average 이용

Average case는 row값과 col값의 평균을 bound으로 설정한다.

**def get\_bound**(*node*, *w*)**:** path **=** []  
 **for** i **in** *node*.path**:** *# because of index* path.append(i **-** 1)  
 bound **=** 0  
 **for** i **in** range(0, len(path) **-** 1)**:** bound **=** bound **+** *w*[path[i]][path[i **+** 1]]  
 *# if the edge is the last of node* min\_num **= -**1 *# We assume -1 is infinite*  
 **for** i **in** range(0, len(*w*))**:  
 if** i **not in** path**:  
 if** min\_num **== -**1**:** min\_num **=** *w*[path[**-**1]][i]  
 **elif** min\_num **>** *w*[path[**-**1]][i]**:** min\_num **=** *w*[path[**-**1]][i]  
 bound **=** bound **+** min\_num **/** 2  
 *# if the edge is not last of node* **for** i **in** range(0, len(*w*))**:  
 if** i **not in** path**:** min\_num1 **= -**1  
 **for** j **in** range(0, len(*w*))**:  
 if** j **not in** path[1**:**] **and** j **!=** i**:  
 if** min\_num1 **== -**1**:** min\_num1 **=** *w*[i][j]  
 **elif** min\_num1 **>** *w*[i][j]**:** min\_num1 **=** *w*[i][j]  
 min\_num2 **= -**1  
 **for** j **in** range(0, len(*w*))**:  
 if** j **not in** path[1**:-**1] **and** j **!=** i**:  
 if** min\_num2 **== -**1**:** min\_num2 **=** *w*[j][i]  
 **elif** min\_num2 **>** *w*[j][i]**:** min\_num2 **=** *w*[j][i]  
 bound **=** bound **+** (min\_num1 **+** min\_num2) **/** 2  
 **return** bound

● Bound: col 이용

Col case는 col 값을 이용하여 min\_num을 구하고 path 구성 시 뒤에서부터 구성한다.

( Node.path **=** [number not in path] **+** Node.path )

**def get\_bound**(*node*, *w*)**:** path **=** []  
 **for** i **in** *node*.path**:** *# because of index* path.append(i **-** 1)  
 bound **=** 0  
 **for** i **in** range(0, len(path) **-** 1)**:** bound **=** bound **+** *w*[path[i]][path[i **+** 1]]  
 *# if the edge is the last of node* min\_num **= -**1  
 **for** i **in** range(0, len(*w*))**:  
 if** i **not in** path**:  
 if** min\_num **== -**1**:** min\_num **=** *w*[i][path[0]]  
 **elif** min\_num **>** *w*[i][path[0]]**:** min\_num **=** *w*[i][path[0]]  
 bound **=** bound **+** min\_num  
 *# if the edge is not last of node* **for** j **in** range(0, len(*w*))**:  
 if** j **not in** path**:** min\_num **= -**1  
 **for** i **in** range(0, len(*w*))**:  
 if** i **not in** path[**:-**1] **and** i **!=** j**:  
 if** min\_num **== -**1**:** min\_num **=** *w*[i][j]  
 **elif** min\_num **>** *w*[i][j]**:** min\_num **=** *w*[i][j]  
 bound **=** bound **+** min\_num  
 **return** bound

결과:

avarage의 경우 들어오는 값과 나가는 값의 최적값을 계산하여 평균하는 것이기 때문에 bound의 값은 항상 row로 구한 것보다 작게 된다. 따라서 더 많은 노드 연산이 필요하다.

col의 경우 row를 구한 것과 반대로 path를 뒤에서부터 탐색한 것과 결과가 같다. 따라서 row와 걸린 시간이 비슷하다.

위의 그래프의 결과에 따라 average case는 실제 Best Search 알고리즘에 적용시 비효율성이 예상된다, col version의 경우에는 기존의 row를 사용하는 Best Search과 비슷한 time complexity를 나타내므로 input matrix에 따라 col이 대체로 사용 될 수 있다.