알고리즘분석

**HW#56 Report**

2011147068 김정환

2011147115 허재화

2012147562 최인호

1. **목차**
   1. **문제 분석**
   2. **각 알고리즘에 대한 보고서**
      1. **DP**
      2. **Backtracking**
      3. **BST**
      4. **Genetic**
      5. **Annealing**
   3. **결과분석 및 정리**
2. **문제 분석**

* TSP 문제를 DP, Backtracking, BST, Genetic, Annealing 5가지 방법으로 해결하고, 그 결과를 분석한다..
* 주어진 Input 파일을 이용해 vertex와 edge를 표현한다.
  + Vertex는 각각의 좌표를 가진다.
  + Edge는 인접행렬로 표현하며, 각 인접행렬의 row, col 값은 Vertex[row]와 Vertex[col] 사이의 거리를 가진다.

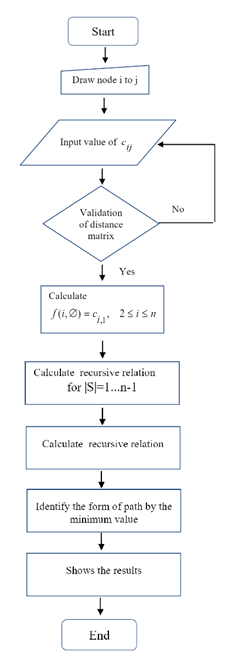
1. **Solving TSP**

**알고리즘 : DP**

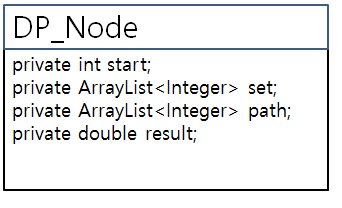
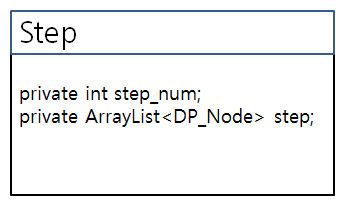
* + 1. 어떤 Vertex i에서 Vertex의 집합을 거쳐 다시 1로 들어오는 방법을 f(i, Set) 이라고 하자.
    2. 를 Edge(I, j)의 값이라고 하고 , S는 거치는 vertex의 집합이라고 할 때 DP 식은 아래와 같다.
    3. ) =
    4. V가 vertex의 집합이라고 할 때, 1번 vertex에서 출발하여 모든 vertex를 다 돌아 1로 다시 들어 오는 값은 다음과 같다.

=

* + 1. 즉, i에서 출발하여 아무 vertex도 거치지 않고 1번 vertex로 가는 방법 수가 된다.
    2. V가 {1, 2, 3, 4}라고 할 때, 예는 다음과 같다,
    3. 의 값이 가장 작다고 하면 최종 경로는 [1->3->2->4->1]가 된다.
    4. S의 크기에 대해 DP 점화식을 적용 시킬 수 있다,
  1. **Flow chart**

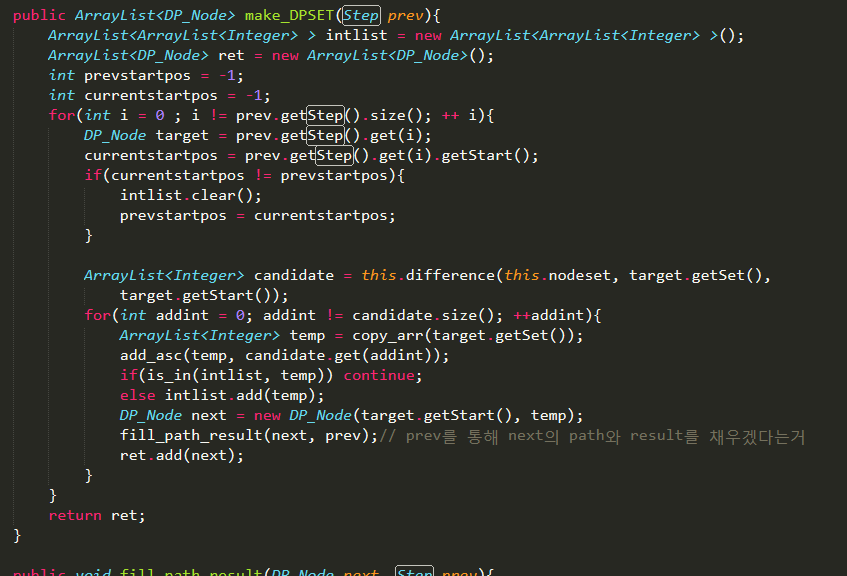


1. **구현**
2. **Class**

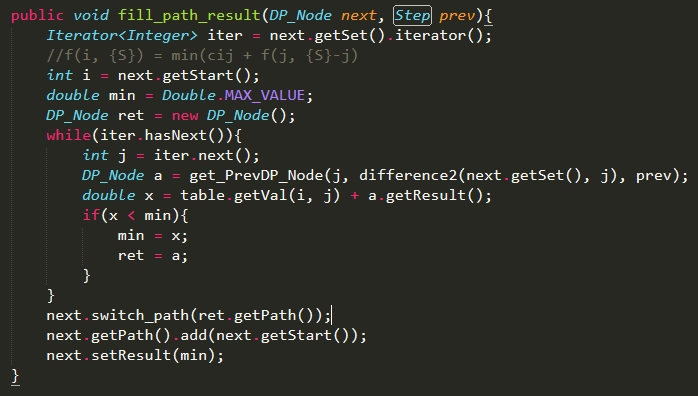


* **Step** 
  + Vertex의 집합 S의 크기가 같은 것들을 모은 것이다.
  + Vertex가 총 4개라고 할 때 Step(2)는  가진다.
  + Verte가 총 n개일 때, Step(n-2)이면 Memoization을 끝내고 Step(n-2)를 이용해 최종 를 구할 수 있다.
* **DP\_Node**
  + 하나의 f의 정보를 표현하기 위해 사용했다.
  + 예를 들어를 표현하면, start는 2, set는 {3,4}, path는 {4,3}(경로가 4->3일때) 와 이전 step에서 구한 최소 dp값을 result에 가지고 있다.

1. **핵심 구현**
   * 1. S의 크기가 N-1인 Step을 통해 크기가 N인 DP\_Node 들을 모두 구해 next Step으로 지정한다..
     2. Make\_DPSET이란 함수에서 크게 도는데, prev의 값을 모두 iteration하며 현재 prev의 DP\_Node에서 만들 수 있는 next DP\_Node를 만들어나가는 역할을 한다.



* 이 함수는 이전 S의 집합이 n개인 Step을 보고 S의 집합이 n+1개인 Step의 DP\_Node set를 구하는 함수이다..
* Prev가 일 때, output은 , 이 된다.
* 과 를 보며 를 만드는 식으로 구현을 했다. 각각에 대해 를 만든뒤, result들을 비교해서 가장 result가 작은 를 선택해 next Step에 넣는다.



* 위의 코드가 이전 단계를 수행하는 코드이다.

1. **분석**

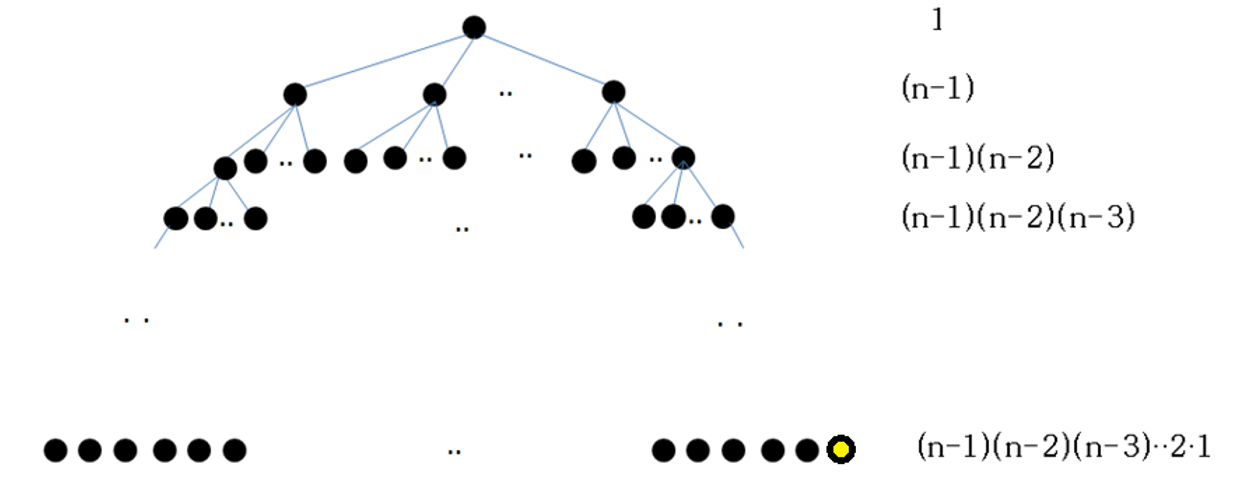
* Dynamic programming을 이용해 TSP를 푸는 방법의 Big Oh notation은 다음과 같다,
* 일 때, 를 구하는데 걸리는 space complexity는 이 된다. 를 알기 위해서는 의 값을 모두 알고 있어야 된다. 이 식은 다시 말해서, start값에 들어갈 수 있는 값이 총 2부터 n까지이므로 n-1이 되고, S는 전체 V에서 1과 start값을 뺀 집합에서 i개를 선택하는 경우의 수가 되기 때문이다.
* 결과적으로을 구하는 것은 이므로 n개의 node를 가진 tsp에 필요한 data들만 해도 )이 된다.
* 여기에 Set의 개수가 i인 Step을 구하는 알고리즘은 다음과 같다. 다음 step의 1개의 DP\_Node를 구하는 데는, 개의 이전 step의 DP\_Node에서 최소 result값을 구하는 과정이 수반 되므로 O() 이고 한 step은 O()개의 DP\_Node로 이뤄 지므로 최종 시간 복잡도는 아래와 같다.
* 이 나오게 된다.

1. **구현과정에서 겪은 어려움**

* 를 살펴보는 문제를 어떻게 줄일 수 있을까 해서 Hash를 사용해봤지만 그리 큰 소용이 없었다.
* 라도 줄이기 위해 ,prev Step과 next Step만을 가지게 하였다. 그러나 그 과정에서 생기는 cost가 많아 DP 의 Performance가 너무 느렸다.

**알고리즘 : Backtracking**

Backtracking은 기본적으로 State Space Tree를 이용해 가능한 모든 경우를 검사하여 최적해를 찾는 알고리즘이다. TSP의 경우 시작 vertex를 root로하여 가능한 모든 경로들을 트리로 찾고, 각각의 경우에 대해 경로의 값을 구하여 비교한다.



TSP를 수행했을 때, State Space Tree는 위의 그림과 같이 형성된다. 그래서 실행시간은 vertex수 n에대하여 exponential하게 걸린다.

이 알고리즘을 이용하여 TSP문제는 푼 과정은 다음과 같다.

State Space Tree의 각 노드는, 각각의 tour와 curDist, 그리고 vlist를 갖는다. tour는 그 시점에서 TSP가 지나온 vertex들을 담고있다. 그리고 curDist는 그 tour경로의 모든 edge값의 합이다. vlist는 그래프의 vertex들을 저장하고 있는 list로서 각 vertex의 flag데이터를 이용하여 특정 vertex가 이미 도달한 vertex인지 검사하는데 사용한다. 우리는 Backtracking기법은 재귀적으로 구현하였다. 그래서 매번 BacktrackTSP가 실행될 때마다 현재 tour가 그래프의 총 vertex개수만큼의 크기를 갖는지 검사한다. 그렇다면 그 경로는 complete한 하나의 경로이므로 그 경로의 길이를 구해서, 이전까지 나온 경로의 최소값과 비교하여 값과 최종경로를 갱신한다. 현재 tour가 그래프의 총 vertex개수만큼의 크기를 갖지 못한다면, 아직 그 tour가 도달하지 못한 vertex들을 하나씩 전부 추가하여 새로운 subproblem을 만들고, 그 subproblem들은 State Space Tree상에서 그 노드의 자식노드들이 된다. 그 자식노드들의 curDist값은 부모노드의 curDist값에다가 새로운 vertex를 추가하기 위해 이어지는 edge의 값을 더한값이 된다. 그리고 Backtracking의 구현 과정에서 우리는 재귀함수를 사용하며, DFS(Depth-First-Search)방식을 사용한다. 이렇게 거의 모든 State Space Tree를 탐색하지만, 만약에 Tree를 형성하는 중간과정에서의 curDist값이 지금까지 구해진 TSP경로의 최소값보다 크다면, 그 경로로부터는 subproblem을 더 만들지 않고, Tree상에서 더 이상 자식노드를 형성하지 않는다. 그 이유는 문제에서 주어지는 그래프는 각 edge의 값이 0보다 크며, 따라서 남은 vertex를 추가하는 과정에서 dist값이 더 작아지는 경우는 기대할 수 없기 때문이다. 이렇게 모든 State Space Tree를 탐색하고 나면 최적의 TSP경로와 그 값이 구해진다.

**알고리즘 : Branch And Bound**

BranchAndBound는 BackTracking 알고리즘과 같이 State Space Tree를 구현하여 문제를 해결한다. 따라서 최악의 경우에 Backtracking과 마찬가지로 vertex수 n에 대하여 exponential한 시간이 걸린다. 하지만 실제로는 좋은 설계에 의해 전형적인 경우에 대하여 매우 효율적인 시간에 최적해를 구할 수 있다. 이 알고리즘은 보통 최적해 구하기(Optimization problem)에 적용될 수 있다. Backtracking기법과의 차이점은, 각 subproblem을 구할 때마다, 그 subproblem이 유망한지 여부를 결정하기 위해 Bound를 계산하고, 그 Bound값이 지금까지 찾은 최적해보다 좋지 않으면 State Space Tree의 그쪽 가지를 잘라내는 방식을 사용한다는 점이다. 여기서 Bound란, 그 subproblem으로부터 가지를 뻗어나갔을 때(branch) 얻을 수 있는 최적해의 한계를 의미한다. 그리고 Backtracking에서 BFS나 DFS를 사용했던 것과는 달리 Branch And Bound에서는 lowerBound에 대한 Best-First-Search를 사용한다. Best-First-Search란 State Space Tree에서 다음 진행할 subproblem을 선택하는데 있어서 lowerBound값이 가장 유리한(최소값 찾는 문제에선 가장 작은)것을 선택해나가는 것이다. 그리고 complete path가 구해져서 어떤 TSP경로값이 구해졌을 때, 그 값보다 큰 lowerBound를 갖는 subproblem들은 Tree에서 더 이상 subproblem을 만들지 않는다. 이게 가지치기(branch) 개념이다. State Space Tree에 있는 앞으로 확인해야 할 subproblem들은 전부 하나의 ArrayList에 저장해 놓았다. 위에서 얘기한 것처럼 lowerBound가 현재 구해진 최소값보다 작은 것 중에서 가장 작은 것을 찾고 그것으로부터 또 subproblem을 생성하는 과정을, 현재 구해진 최소값보다 lowerBound가 작은 subproblem이 없거나, 완전히 subproblem이 없을 때까지 반복한다. 그러면 그 때까지 구해진 complete path distance중에 가장 작은값이 전체 최적값으로 update되어 있을 것이고, 그때의 경로가 최종경로로 저장되어 있을 것이다.

여기서 중요한 과정으로 lowerBound를 구하는 과정이 있다. 이 알고리즘에서 가장 중요한 부분이라고 할 수 있으며, 이 lowerBound를 어떻게 설정하느냐에 따라 이 알고리즘의 효율성이 결정된다. lowerBound를 별 의미없는 값으로 정하면 가지치기가 제대로 이루어지지 않기 때문이다. TSP문제에서의 lowerBound를 구하는 식은 다음과 같다.

그래프의 전체 vertex집합 : V

어떤 subproblem의 현재까지 도달한 vertex 집합 : S

그 subproblem의 현재 경로

: a - ~~~ - b (a에서 시작하여 b를 추가하여 만들어진 subproblem)

이렇다고 했을 때,

lowerBound = MST(V-S) + dist(a - ~~~ - b) + MIN(a와 V-S를 잇는 edge)+MIN(b와 V-S를 잇는 edge).

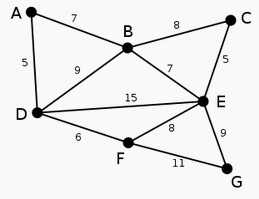
* + - * dist(a - ~~~ - b)는 이 subproblem의 partialTour의 distance.

여기서 MST는 여러가지 구하는 방식이 있다. 우리는 프림 알고리즘(Prim Algorithm)을 사용하였다. 프림 알고리즘은 다음과 같다.

**[프림 알고리즘]**

그래프의 한 vertex에서 시작하여, 트리가 모든 vertex를 포함할 때까지 다음 과정을 계속한다.

(선택된 vertex를 primTour에 넣는다. 그리고 그 primTour에 존재하는 vertex들 각각에서, 아직 선택되지 않은 vertex로 이어지는 모든 edge들 중에 가장 작은 가중치를 갖는 edge를 찾는다. 최소값이 여러 번 나타날 경우 아무거나 선택한다. 그리고 그 edge의 값을 MST값에 더하면서, 그 edge로 인해 새로 연결되는 vertex는 primTour에 추가한다.)

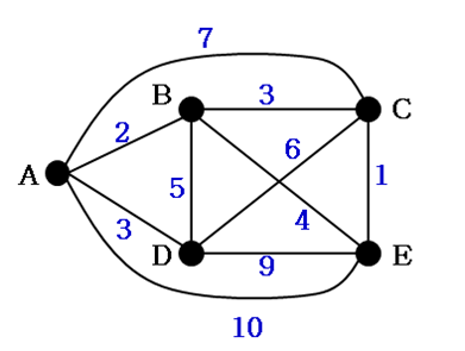


위 그래프에 대해서 프림 알고리즘의 실행순서를 보면 다음과 같다.

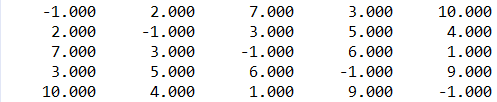
처음에 A를 primTour시작점으로 선택한다고 하자. 그러면 A에서 아직 선택되지 않은 나머지 vertex들로 이어지는 edge중에서 가장 작은 edge는 edge(A, D)=5 이다. 그러면 D가 primTour에 추가되고 MST값에는 5가 더해진다. 그리고 반복해서, A나 D에서 그 둘을 제외한 나머지 vertex들로 이어지는 edge들(AB, DB, DE, DF)중 가중치가 가장 작은값은 edge(D, F)=6이다. F는 primTour에 추가되고 MST값에는 6이 더해진다. 이런 과정을 primTour에 추가되지 않은 vertex가 없을 때까지 반복하면 MST가 구해진다. 우리는 이 과정에서 하나의 vertex를 primTour에 추가할 때마다 primTour의 size만큼, V개의 vertex에대한 경로를 전부 검사하고, 그 과정을 primTour의 size가 V가 될 때까지 반복하므로 O(V^3)의 연산을 수행한다.

Backtracking 알고리즘과 BranchAndBound알고리즘은 비슷한 부분이 많다. Backtracking기법을 좀더 개선하여(브랜치, lowerBound개념 추가) 과정의 효율성을 높이면 BranchAndBound와 같은 알고리즘을 생각할 수 있다.

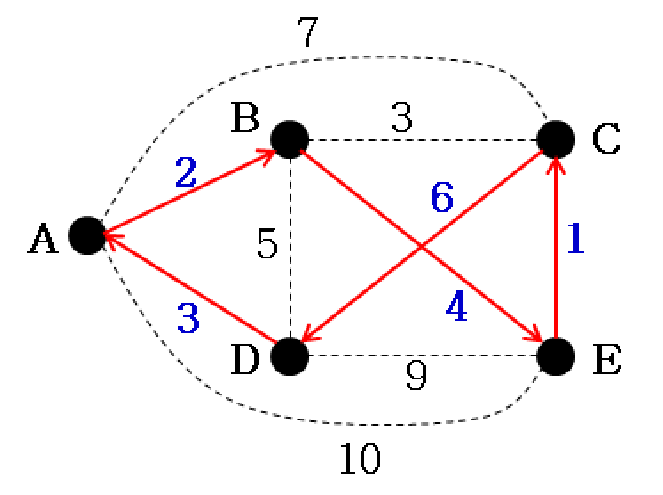
그래서 위의 두가지 알고리즘에 대해서 같은 예제(강의노트 예제)를 돌려보면서 비교를 해보았다.

**

이 예제의 각 vertex들과 edge에 대하여 인접행렬을 표현하면 다음과 같다.



\*행 번호와 열 번호가 같은 곳에는 -1을 넣어 edge가 없음을 표현.



우선 이 문제의 최종 TSP경로를 나타내면 위와 같고, 그때의 distance는 16이다.

**-Backtraking**

이 알고리즘을 사용했을 때 우리가 돌려본 프로그램에서 만들어지는 subproblem들을 보면 다음과 같다.

[tour] [curDist] [minDist]

partialTour : [0] curDist(현재 distance) : 0.0

partialTour : [0, 1] curDist(현재 distance) : 2.0

partialTour : [0, 1, 2] curDist(현재 distance) : 5.0

partialTour : [0, 1, 2, 3] curDist(현재 distance) : 11.0

partialTour : [0, 1, 2, 3, 4] curDist(현재 distance) : 20.0 -> 30으로 갱신

partialTour : [0, 1, 2, 4] curDist(현재 distance) : 6.0

partialTour : [0, 1, 2, 4, 3] curDist(현재 distance) : 15.0 -> 18로 갱신

partialTour : [0, 1, 3] curDist(현재 distance) : 7.0

partialTour : [0, 1, 3, 2] curDist(현재 distance) : 13.0

partialTour : [0, 1, 3, 2, 4] curDist(현재 distance) : 14.0

partialTour : [0, 1, 3, 4] curDist(현재 distance) : 16.0

partialTour : [0, 1, 3, 4, 2] curDist(현재 distance) : 17.0

partialTour : [0, 1, 4] curDist(현재 distance) : 6.0

partialTour : [0, 1, 4, 2] curDist(현재 distance) : 7.0

partialTour : [0, 1, 4, 2, 3] curDist(현재 distance) : 13.0 -> 16으로 갱신

partialTour : [0, 1, 4, 3] curDist(현재 distance) : 15.0

partialTour : [0, 2] curDist(현재 distance) : 7.0

partialTour : [0, 2, 1] curDist(현재 distance) : 10.0

partialTour : [0, 2, 1, 3] curDist(현재 distance) : 15.0

partialTour : [0, 2, 1, 4] curDist(현재 distance) : 14.0

partialTour : [0, 2, 3] curDist(현재 distance) : 13.0

partialTour : [0, 2, 4] curDist(현재 distance) : 8.0

partialTour : [0, 2, 4, 1] curDist(현재 distance) : 12.0

partialTour : [0, 3] curDist(현재 distance) : 3.0

partialTour : [0, 3, 1] curDist(현재 distance) : 8.0

partialTour : [0, 3, 1, 2] curDist(현재 distance) : 11.0

partialTour : [0, 3, 1, 2, 4] curDist(현재 distance) : 12.0

partialTour : [0, 3, 1, 4] curDist(현재 distance) : 12.0

partialTour : [0, 3, 1, 4, 2] curDist(현재 distance) : 13.0

partialTour : [0, 3, 2] curDist(현재 distance) : 9.0

partialTour : [0, 3, 2, 1] curDist(현재 distance) : 12.0

partialTour : [0, 3, 2, 4] curDist(현재 distance) : 10.0

partialTour : [0, 3, 2, 4, 1] curDist(현재 distance) : 14.0

partialTour : [0, 3, 4] curDist(현재 distance) : 12.0

partialTour : [0, 3, 4, 2] curDist(현재 distance) : 13.0

partialTour : [0, 4] curDist(현재 distance) : 10.0

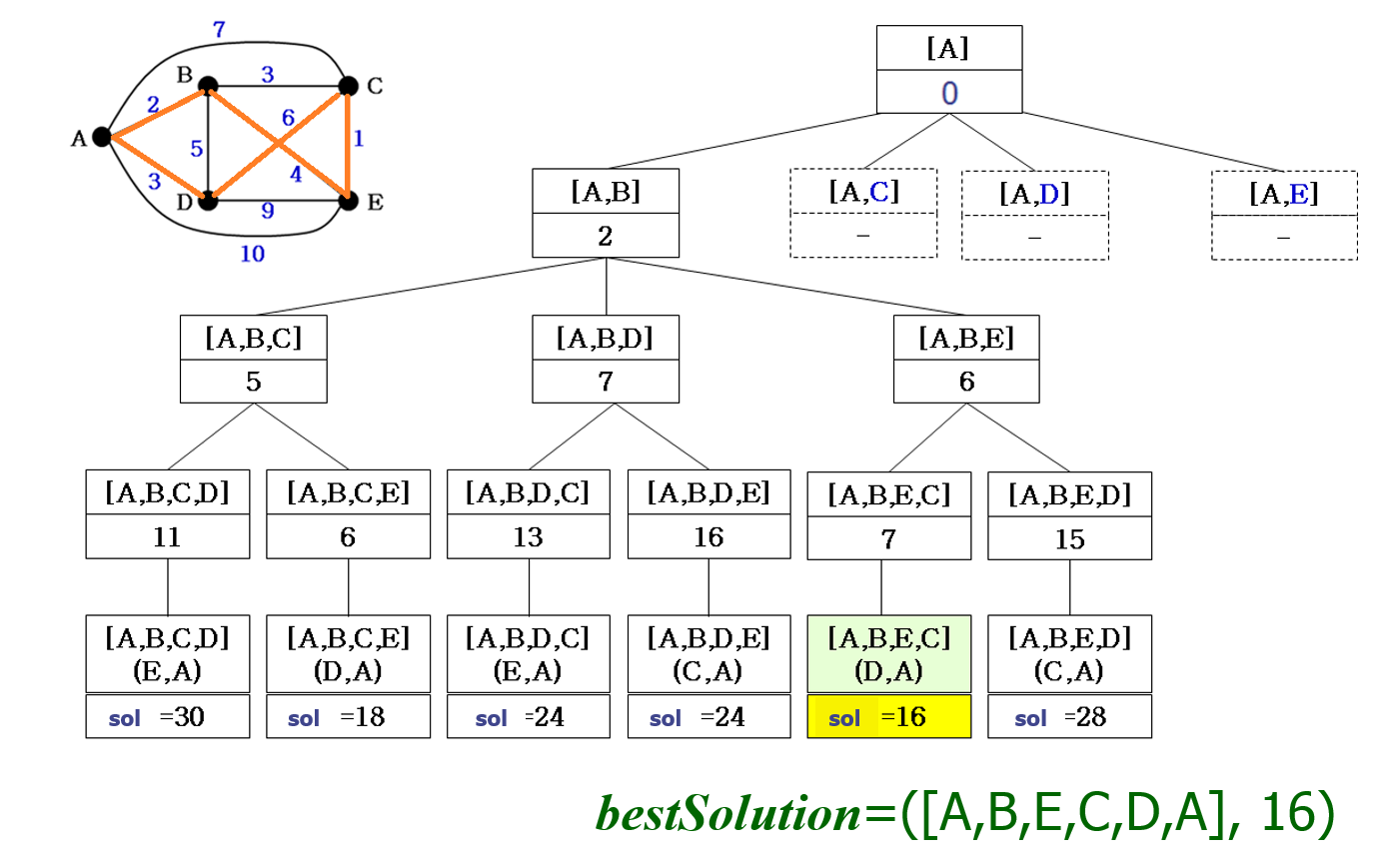
partialTour : [0, 4, 1] curDist(현재 distance) : 14.0

partialTour : [0, 4, 2] curDist(현재 distance) : 11.0

partialTour : [0, 4, 2, 1] curDist(현재 distance) : 14.0

위와 같은 순서대로 Backtracking이 이루어지고 있음을 확인할 수 있다. 위에서 생각한 것처럼 거의 가능한 모든 경로를 전부 검사하고 있으며, 중간에 몇몇 가능한 경로들을 건너 뛰는 것은, 그 경로들의 curDist값이 그전에 구해진 minDist값보다 커서 그렇다는 것을 알 수 있었다.

State Space Tree는 위의 예제에 대해서 강의노트에서 보였던 것과 같게 나온다.



**-BranchAndBound**

같은 예제 그래프에 대하여 BranchAndBound를 적용하여 보았다. 이번에는 각 subproblem에 대해 partialTour와 lowerBound를 확인해 보았다.

]

[tour] [lowerBound] [minDist]

partialTour : [0, 1] lowerBound : 15.0

partialTour : [0, 2] lowerBound : 19.0

partialTour : [0, 3] lowerBound : 14.0

partialTour : [0, 4] lowerBound : 21.0

partialTour : [0, 3, 1] lowerBound : 19.0

partialTour : [0, 3, 2] lowerBound : 16.0

partialTour : [0, 3, 4] lowerBound : 18.0

partialTour : [0, 1, 2] lowerBound : 18.0

partialTour : [0, 1, 3] lowerBound : 21.0

partialTour : [0, 1, 4] lowerBound : 16.0

partialTour : [0, 1, 4, 2] lowerBound : 16.0

partialTour : [0, 1, 4, 3] lowerBound : 28.0

여기서 subproblem으로 partialTour : [0, 1, 4, 2] lowerBound : 16.0 이 선택되면서(lowerBound가 16으로 남아있는 subproblem중에 가장 작다) minDist값이 16으로 갱신된다.

* 16으로 갱신.

partialTour : [0, 3, 2, 1] lowerBound : 26.0

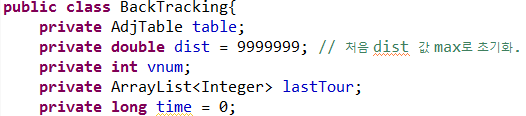
partialTour : [0, 3, 2, 4] lowerBound : 16.0

* + - * 남아있는 subproblem중에 lowerBound가 16보다 작은 것은 없으므로 종료된다.

위에서 생각한 것처럼 Best-First-Search가 잘 돌아가고 있으며, lowerBound도 생각한 것처럼 잘 구해지고 있음을 확인할 수 있었다. 그리고 Backtracking과 비교했을 때, vertex가 5개밖에 안되는 작은 크기의 instance임에도 그 subproblem의 개수가 훨씬 적고, State Space Tree탐색이 훨씬 효율적이라는 것을 알 수 있었다.

1. **구현**

**[Backtracking]**



BackTracking 클래스의 멤버변수들.

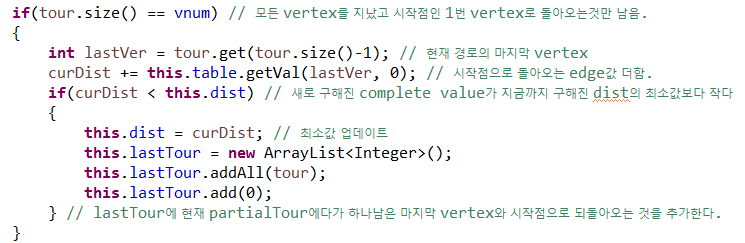
table : 그래프의 인접행렬을 저장한다.

dist : TSP경로값을 갱신할 변수이며, 알고리즘이 끝난 후에는 최적의 TSP경로값이 저장된다.

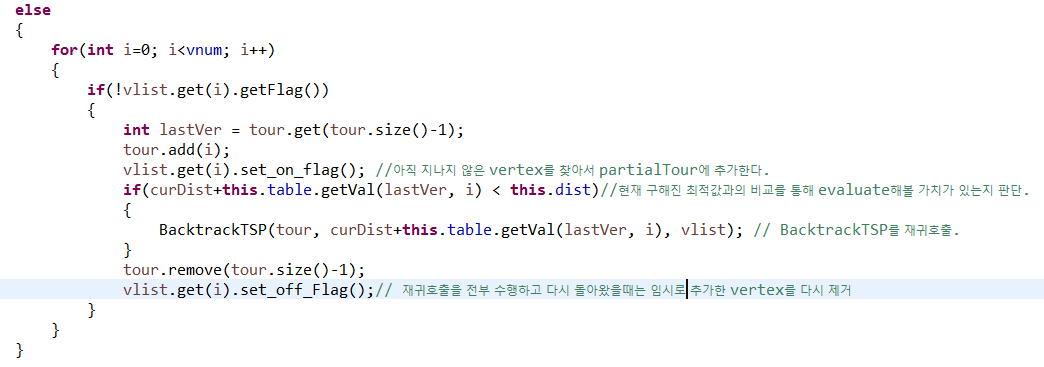
vnum : vertex의 개수를 저장하는 변수

lastTour : TSP경로값이 갱신될 때마다 그 경로로 같이 갱신될 변수이며, 마지막에는 최적의 TSP경로가 저장된다.

time : 알고리즘 수행 시간을 측정하기 위한 변수.



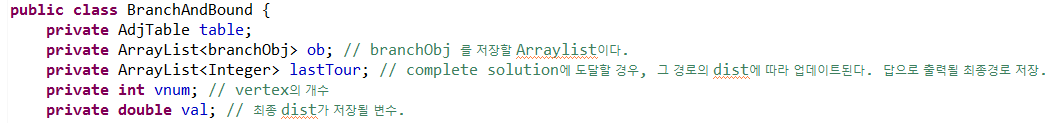
BacktrackTSP가 호출되면 일단 현재 tour의 size를 검사하여 모든 vertex를 지났는지 확인한다. 모든 vertex를 지났다는 것은 하나의 complete path가 완성되었다는 뜻이므로, 그 경로의 distance를 구하여 현재의 최적값과 비교를 해봐야 한다. 일단 그 partialTour의 distance값은 curDist에 저장되어 있으므로, 그 값에다가 마지막 vertex에서 시작점으로 다시 돌아오는 값만 더하여 준다. 그리고 그 값을 현재의 최적값(dist)값과 비교하여 더 작으면 dist값을 update해주면서 lastTour도 그 partialTour로 바꿔준다. 그리고 마지막에 시작점으로 되돌아오는 것을 추가해준다.



현재 tour의 size가 vertex의 총 수보다 작은 경우, 아직은 complete path가 만들어지지 않은 것이다. 이런 경우에는 이 subproblem으로부터 새로운 subproblem들을 만들어야 한다. 그래서 모든 vertex들을 돌면서 이 vertex가 지나온 점인지 확인하고, 안 지나온 vertex들을 하나씩 추가해서 새로운 subproblem으로 BacktrackTSP를 재귀호출한다. 여기서 중요한 점은, BacktrackTSP함수를 재귀호출하기 전에 i vertex를 추가하여 호출하고, 그 재귀호출이 끝나고 돌아오면 i vertex를 다시 제거해줘야 한다는 점이다. 이유는 i vertex는 그다음 재귀호출 때 새로운 subproblem을 생성하는 과정에서 추가된 vertex이고 다시 돌아왔을 때는 i vertex가 추가되기 이전의 subproblem으로 되돌아 와야하기 때문이다.

위의 BacktrackTSP 재귀호출을 전부 완료하고 나면, dist에는 TSP경로의 최소값이, lastTour에는 그 경로가 저장되어 있다.

**[BranchAndBound]**



BranchAndBound 클래스의 멤버변수들

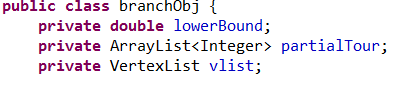
table : 그래프의 인접행렬을 저장한다.

ob : branchObj의 Arraylist. branchObj 는 알고리즘 중간과정에서 생기는 subproblem들을 담을 class이다.

lastTour : TSP경로값이 갱신될 때마다 그 경로로 같이 갱신될 변수이며, 마지막에는 최적의 TSP경로가 저장된다.

vnum : vertex의 개수를 저장하는 변수

val : TSP경로값을 갱신할 변수이며, 알고리즘이 끝난 후에는 최적의 TSP경로값이 저장된다.



branchObj 클래스의 멤버변수들

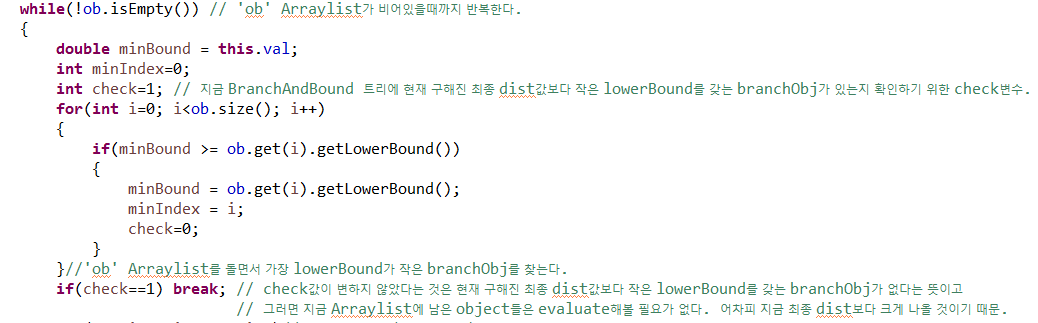
lowerBound : 그 subproblem의 lowerBound

partialTour : 그 subproblem이 지금까지 지나온 path 저장.

vlist : 지나온 vertex 정보 저장.

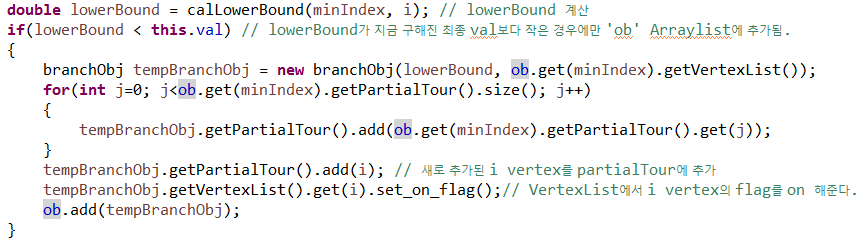


BranchAndBound의 생성자이다. tour라는 임시 path 저장 변수를 선언하여, 시작 vertex를 추가해준다. 그리고 val값은 처음에 9999999라는 값을 넣어서 더 작은 값이 나오면 갱신하도록 한다. ob는 branchObj로 이루어진 새로운 ArrayList를 선언한다. 그리고 ob의 첫번째 subproblem을 추가하기 위해 tempBranchObj를 만들어서 아까 만들어 두었던 tour를 partialTour로 지정해주고 ob에 추가시킨다.

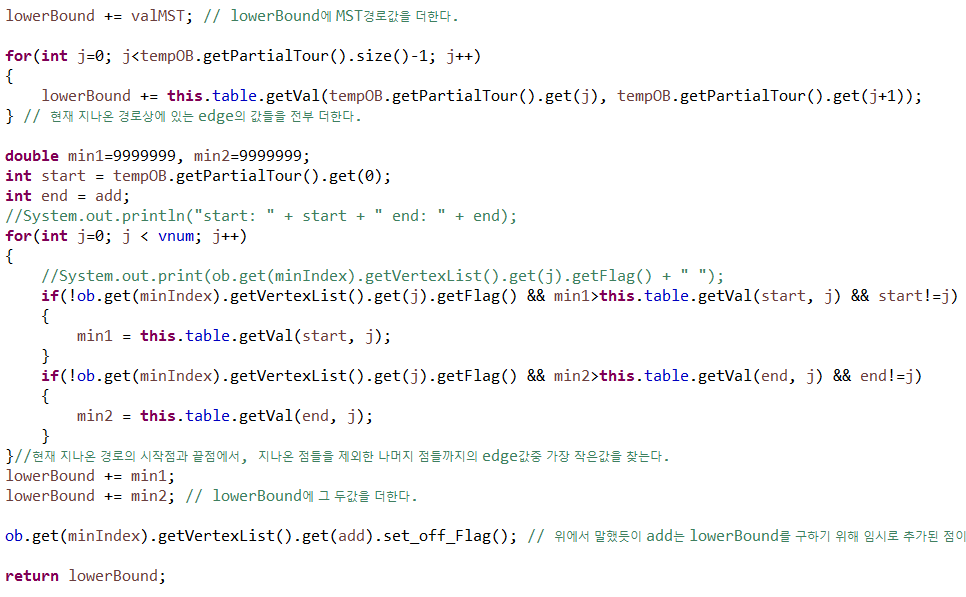


ob 리스트에 더 이상 남은 branchObj가 없을 때까지 다음 과정을 반복한다.

일단 ob 리스트 안에서 lowerBound가 가장 작은 subproblem을 찾는다. 여기서 조건은 지금까지 구해진 최적값보다 lowerBound가 작아야 한다는 점이다. 그 이유는 이미 나온 최소값보다 lowerBound가 크면, 그쪽 branch를 아무리 탐색해봐야 그 lowerBound보다는 크거나 같은 경로값이 나올 것이고, 그것은 이미 구해진 dist보다 크다는 뜻이므로 의미가 없다. 그래서 그런 subproblem은 더 이상 탐색하지 않고 가지치기 해준다. 이게 BranchAndBound의 중요한 과정이다. 만약 ob 리스트 안에 현재 dist값보다 작은 lowerBound를 갖는 branchObj가 존재하지 않으면, 더 이상 탐색해볼 가치가 있는 subproblem이 없다는 뜻이므로 거기서 알고리즘을 종료하고 그때까지 나온 최적값을 결과로 택한다.



어떤 subproblem(lowerBound가 가장 작은)이 선택되었을 때, 그 subproblem의 tour가 포함하지 않는 vertex i를 찾는다. 그리고 그 vertex i가 추가되었을 때 complete path가 생성되면 그 경로값을 현재 dist값과 비교하여 업데이트 작업을 처리해준다. 물론 partialTour도 같이 업데이트 해준다. 그리고 아직 complete path에 도달하지 못한 경우에는 vertex i를 추가했을 때 lowerBound를 계산한다. 그리고 그 lowerBound가 현재 dist값보다 작을 경우에만 새로운 subproblem으로 인정돼서 ob에 추가된다. lowerBound가dist보다 큰 경우에는 앞에서 얘기한 이유로 더 탐색할 가치가 없는 것이므로 subproblem으로 추가되지 않는다.

lowerBound를 구하는 과정이다. lowerBound는 위에 알고리즘에서 보인 것처럼 다음의 식을 이용하여 구한다.

lowerBound = MST(V-S) + dist(a - ~~~ - b) + MIN(a와 V-S를 잇는 edge)+MIN(b와 V-S를 잇는 edge).

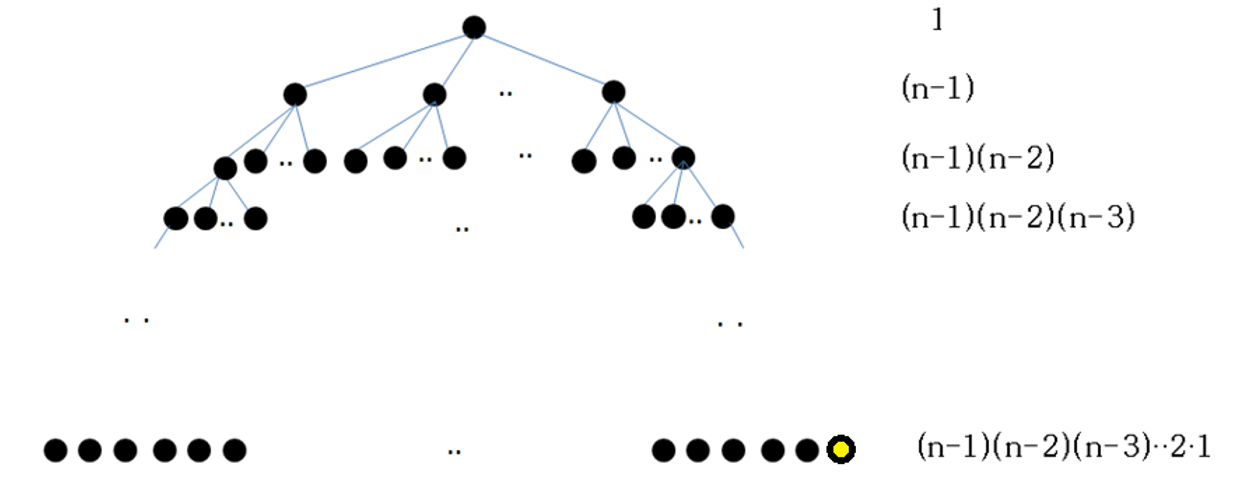
* + - * dist(a - ~~~ - b)는 이 subproblem의 partialTour의 distance.

MST를 구하는 데는 위에서 보인 prim 알고리즘을 사용하였다.

1. **토의**

**Backtracking**

Backtracking은 기본적으로 가능한 모든 경우를 역추적해서 검사하는 기법이다. 따라서 State Space Tree를 그려보면 다음과 같다.



따라서 시간복잡도는

,

그리고 각각의 subproblem에 대하여 모든 vertex들을 검사하여 subproblem을 만드는 작업을 수행하므로 n을 곱한다.

즉 O(n^(n+1))이고 vertex 개수 n에 대하여 exponential한 runtime이 소요되는 것을 알 수 있다.

**BranchAndBound**

BranchAndBound는 Backtracking과 마찬가지로 State Space Tree를 탐색해서 문제를 해결한다. State Space Tree의 총 노드 수는 최악의 경우 Backtracking과 마찬가지로 n에 exponential하게 될 것이다. 그리고 각각의 subproblem에 대하여 lowerBound를 구하기 위해 prim algorithm을 수행하는 과정에서 위에서 보였듯이 O(n^3)의 연산을 수행한다.

즉 최악의 경우 시간복잡도는 Backtracking의 시간복잡도에 n^3을 곱하여 O(n^(n+4))정도가 나온다. (n에 대하여 exponential)

하지만 일반적인 경우에 BranchAndBound는 그보다 훨씬 효율적인 탐색을 한다. 위의 vertex가 5개밖에 없는 예제로 테스트를 해봤을 때 확인할 수 있었던 것처럼 Backtracking보다 훨씬 적은 subproblem만을 확인하며(잘 설계된 lowerBound로 대부분의 subproblem이 가지치기된다), vertex의 개수가 커지면 Backtraking에 비하여 가지치기 되는 subproblem은 더 많아질 것이다. 그래서 실제로 38개 vertex가 있는 예제를 돌려봤을 때도 Backtraking보다 훨씬 빠른 시간에 BranchAndBound알고리즘이 TSP경로를 찾아내는 것을 확인할 수 있었다. BranchAndBound도 시간복잡도 상으로 분명히 n에대하여 exponential한 수행시간을 갖기 때문에 vertex수가 커지면 TSP문제를 해결하는데 너무 많은 시간이 걸려 현실적으로 답을 구하는 것이 불가능 할 수 있지만, 일반적이고 vertex가 그렇게 크지 않은 문제들에 대해서는 Big-O notation과 무관하게 매우 효율적인 문제 해결을 할 수 있음을 확인했다.

1. **구현과정에서 겪은 어려운 점**

**Backtracking**

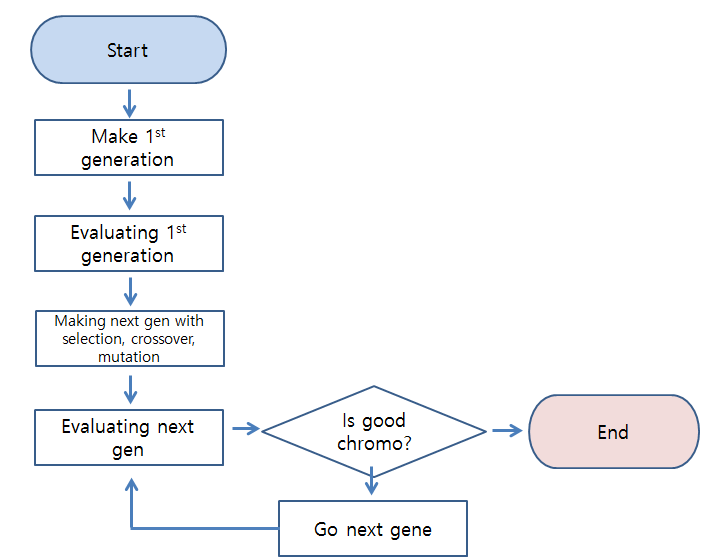
알고리즘의 개념 자체는 크게 어려운 점이 없었다. 간단히 얘기하면 그냥 재귀적으로 모든 경우를 해보고, 단 현재 구해진 최적값과 비교하여 이미 그보다 작은 값이 나올 수 없는 경우에 대해서 pruning하는 것이기 때문이다. 그런데 구현하는 과정에서 각 subproblem들은 partialTour와 vlist(지나온 vertex를 기록하는 flag list)를 가져야 하는데 그 정보들이 subproblem들에서 섞이는 것이 문제가 됐다. JAVA에서 객체를 ‘=’operator을 사용하여 복사하면 그 값만 복사되는 것이 아니라 객체가 가리키는 위치가 같아지기 때문에 다른 subproblem에서 정보를 수정하면 그 정보가 독립적이어야 할 상위 subproblem에까지 영향을 미치기 때문이다. 처음에 이 점을 간과하고 알고리즘을 구현했다가 문제를 발견하고 각 recursive call마다 직전에 정보를 수정한 다음 그 recursive call을 마치고 되돌아오면 수정한 정보를 다시 복원하는 방법으로 해결하였다.

**BranchAndBound**

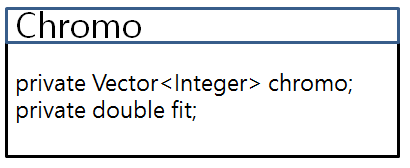
BranchAndBound에서도 Backtracking에서와 비슷한 문제를 겪었다. 두 알고리즘 모두 subproblem들이 각각의 partialTour와 flag 정보(vlist)를 갖는다는 점 때문에, 그 정보가 섞이면 안된다는 중요한 제한이 있다. 그래서 처음에 ‘=’operator로 객체 값 복사를 했던 것으로 인해 잘못된 결과가 나오는 것을 확인하고, 객체를 복사할 때 그 안의 값들을 하나씩 복사하는 방식으로 문제를 해결하였다.

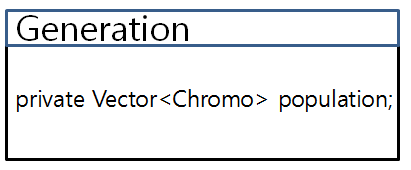
**알고리즘 : GENETIC**

* + 1. 생물이 진화하는데 있어서, 적자 생존의 원칙에 따라 진화를 하되, 이 때 Chromosome의 crossover, mutation을 통해 이전 보다 더 좋은 종으로 진화한다.
    2. 위의 생물 진화 알고리즘을 TSP에 적용하면 단어들은 다음과 같이 정의할 수 있다.
       1. Generation : Chromosome의 집합
       2. Chromosome : TSP를 만족하는 한 path
       3. Selection : 현 세대에서 더 좋은 값들을 선택하는 과정. 이 과정에서는 Tournament-selection을 선택한다.
       4. Crossover : 두 개의 Chromosome이 만나 새로운 Chromosome을 만드는 과정
       5. Mutation : Crossover 이후, 확률적으로 유전자의 정보가 바뀌는 과정, local solution에서 optimal solution쪽으로 찾아갈 수 있는 근거가 된다.
    3. 알고리즘의 전체 흐름은 다음과 같다.
       1. Random pick을 통해 초기 세대를 생성하고 적합도를 평가한다.
       2. 초기 세대에 대한 selection, crossover, mutation을 해 다음 세대를 생성한다.
       3. 만들어진 세대가 목표에 적합한 염색체를 가지면 알고리즘을 종료한다.
       4. 만약 없다면, selection, crossover, mutation을 통해 계속해서 다음 세대를 생성해 다시 3의 과정으로 간다.
    4. **목표 값은 전체 Edge weight의 합에서 n을 나눈 값으로 잡았다.**
    5. Flow chart

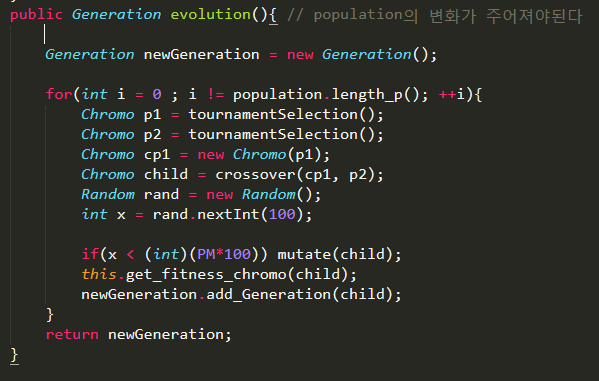


* 1. **구현**
     1. **Class**

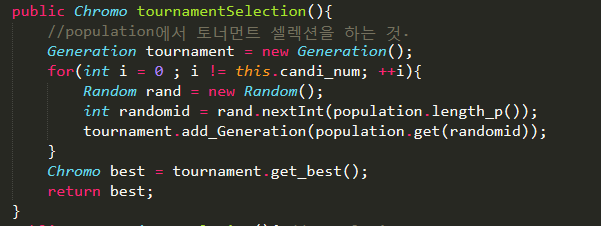




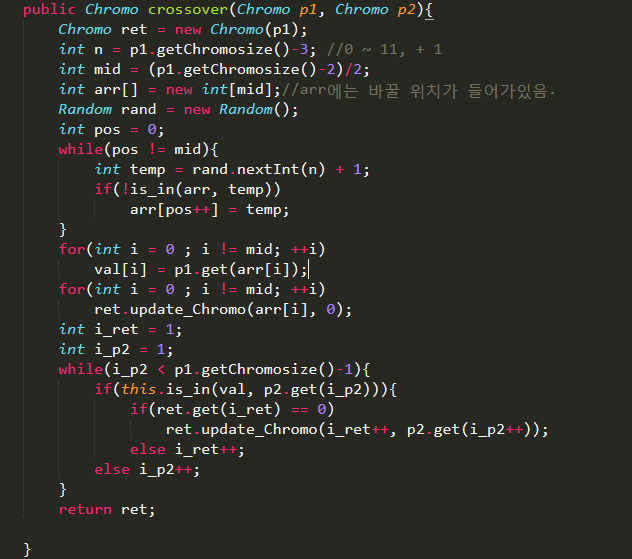
* Chromo
  + 자연 선택 설에서 염색체에 해당하는 부분이다.
  + TSP를 만족하는 path를 가지고 있는 객체이다.
  + Selection, crossover, mutation 모두 Chromo를 리턴한다.
* Generation
  + 자연 선택 설에서 한 세대에 해당하는 부분이다.
  + Chromo의 집합으로 구성된다.
  + next Generation은 current Generation에서 목표치 보다 좋은Chromosome이 없을 시, current gen에서 전이된다.
    1. **핵심 구현**
* make\_first() 함수를 통해 1st generation을 만든다. 그리고 어떤 세대에서 목표 치보다 좋은 염색체가 나올 때 까지, evolution()함수를 통해 다음 세대로 진화한다.
* **Generation evolution()**



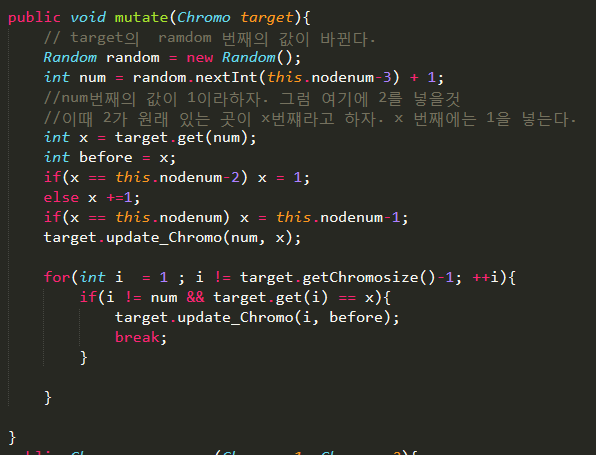
* + Current Generation에서 Next Generation으로 진화시키는 함수이다.
  + Current Generation에서 2개를 tournament selection으로 고른다.
  + 고른 두 개를 crossover 시켜서 child를 만든다.
  + PM에 따라 mutate를 시킨다.
  + 이 과정을, 한 세대에 들어가는 염색체의 수만큼 반복한다.
* **Chromo TournamentSelection()**



* + 한 population에서 random하게 population 수만큼 고른 뒤, 거기서 가장 좋은 Chromosome를 선택한다.(적자생존)
* **Chromo Crossover(Chromo p1, Chromo p2)**



* + 두 Chromosome을 골라 crossover 시켜 새로운 염색체를 만든다.
  + Child에 p1의 값을 복사한다.
  + Child에서 random pick을 (chromosome size/2) 만큼 한다.
  + 그리고 random pick 된 부분의 값들은 p2의 순서대로 채우되, 채워지는 값들은 child에 없는 값들에 한한다.
* **Void mutate(Chromo target)**

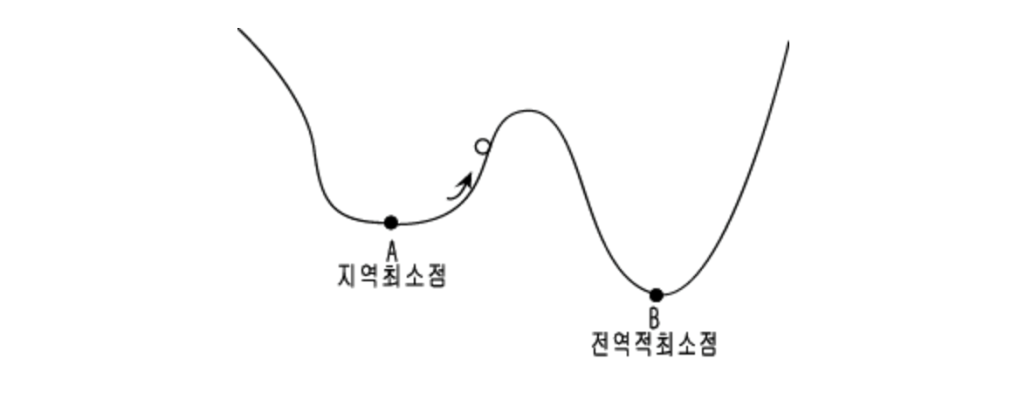


* + Argument로 들어온 target에 대해 mutation을 시킨다.
  + target에서 random하게 2곳의 값을 골라 서로 switching 시킨다.
  1. **분석**
* Genetic algorithm을 사용해 해결하는 Big-Oh notation은 다음과 같다.
* 초기 세대 생성 : 한 Chromo를 만드는데, 1~n사이의 숫자에서 random pick을 하는데 n번, 길이 n번해서 이고 한 population의 수는 n으로 잡았기 때문에 최종적으로 이 된다.
* Tournament Selection : random하게 고른 두 Chromo에서 더 좋은 하나를 구하므로, 선택하는 것에 constant time, random 하게 두 수를 고르는데 이므로 이 된다.
* Crossover : random pick을 mid번 하기 때문에 이고, 각 자리에 넣는 게 이므로, 최종적으로 이 된다.
* Mutation : random pick을 2번 해야 되므로 , 스위칭에 이므로 총 이 된다.
* Evolution : evolution에서 critical한 부분은 crossover이므로, evolution의 한 iteration에는 이된다. 그리고 이 작업을 n번 수행하므로 이 된다.
* 따라서 time complexity는 이 된다.
  1. **구현과정에서 겪은 어려움**
* PPT를 보며 구현을 했는데, Roulette selection과 tournament selection 개념이 혼동 되어 두 개를 합친 selection함수를 구현하였다. 그랬더니 evolution을 아무리 해도 좋아지지가 않았는데 tournament selection을 다시 공부하여 해결했다.

**알고리즘 : Annealing**

Simulated annealing은 커다란 탐색공간에서 주어진 함수의 global optimum 에 대한 훌륭한 근사치를 찾으려고 하는global optimization에 대한 일반적인 확률적 휴리스틱 접근방식이다.

이 알고리즘의 개념은 야금학의 담금질(annealing)에서 따온 것이다. 결정체의 크기를 크게하고 결함(defects)을 작게 하려고 금속에 열을 가하고 냉각시키는 속도를 조절하는(controlled cooling) 기술에 아이디어의 기반을 두고 있다. 열을 가하면 원자는local minimum 에서 떨어져 나가고 더 높은 에너지 상태로 정처없이 방황하게 된다. 서서히 냉각시키면 최초의 상태보다 더 낮은 내부 에너지를 가지는 환경을 찾을 수 있는 기회를 더 많이 가지게 된다.

알고리즘은 해를 반복해 개선함으로써, 현재의 해 근방에 있는 해를 임의로 찾는데, 그때에 주어진 함수의 값과 전역 인자 *T* (온도를 의미한다)가 영향을 준다. 그리고 앞에서 기술한 물리 과정과 비슷한 원리로. *T*(온도)의 값은 서서히 작아진다. 따라서, 처음에는 *T*가 크기 때문에 해가 크게 변화하지만, *T*가 0에 가까워짐에 따라 변화가 줄어든다. 처음은 간단하게 비탈을 올라갈 수 있으므로, Hill climbing으로 문제가 되는 local optimal 빠졌을 때의 대책을 생각할 필요가 없다.

현재 local optimal에 머물러 있더라도 초반 어닐링 과정에서 아래와 같이 표현되는 값에 따라서 변화하는 정도가 T값과 비례하기에 큰 T값인 상태에서 로컬 옵티멀에 빠지지 않고 벗어날 수 있다.

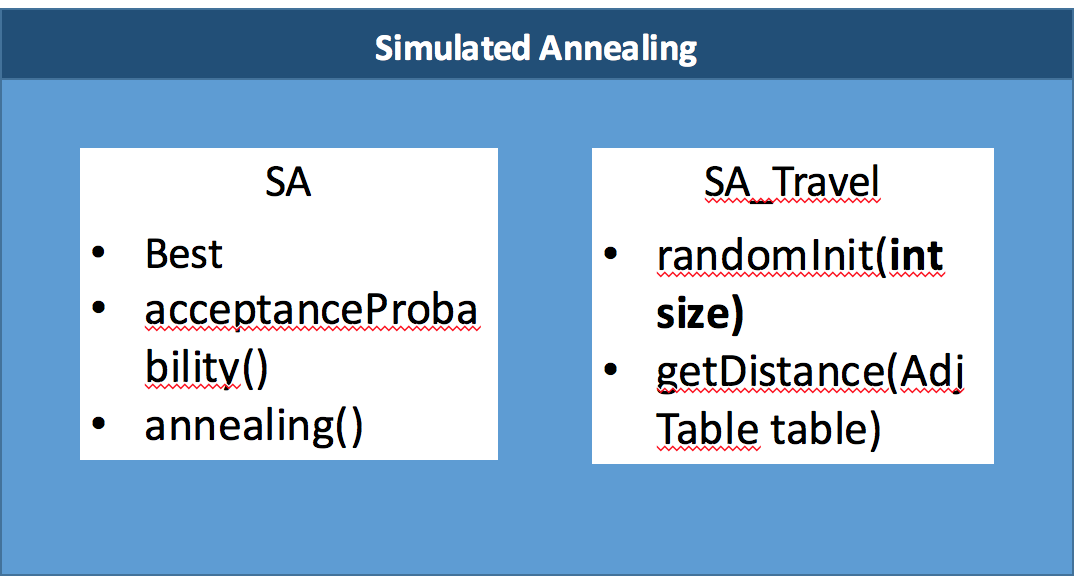


알고리즘은 우선 랜덤 값에서 시작을 하며, 이웃하는 솔루션을 찾고 비교하면서 그 과정에서 발견한 local optimal을 계속해서 추적하는 방법으로 사용된다. 여기서 위의 이미지에서 나오는 수용 확률 만큼 결과값에 구애 받지 않고 이웃 해를 따라 가도록 한다. T값이 단계에 따라 줄어들면서 이웃 해의 결과값에 의존적으로 작용하며, 여러 local optimal들 중 best solution을 찾아낸다.

1. **구현**

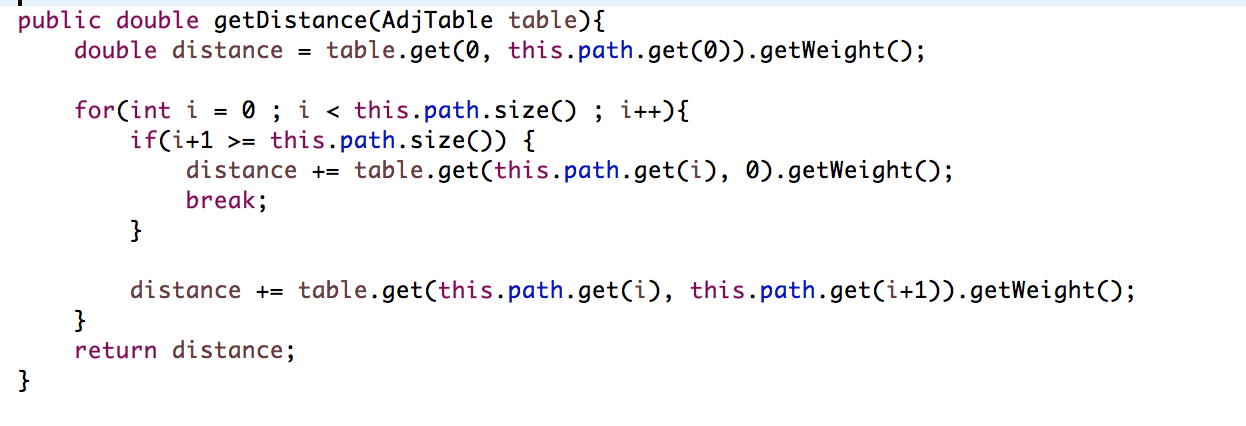
**3.1 Class 설명**

* 프로그램 중 Simulated annealing에서 사용하는 클래스는 SA와 SATravel로 2개가 있다.
* Best는 annealing을 수행하면서 계속해서 로컬 옵티멀을 저장한다.
* acceptanceProbablility는 에너지값과 T를 입력으로 받고 수용 확률값을 출력한다.
* Annealing은 실제 루프를 돌면서 어닐링 과정을 수행한다.
* randomInit은 모든 path를 초기화 하기 위해서 사용한다. 랜덤으로 완성된 경로를 출력하기 때문에 아닐링의 초기 path를 주는데에 사용된다.
* getDistance는 adjTable의 값을 통해서 path의 총 distance를 출력한다.

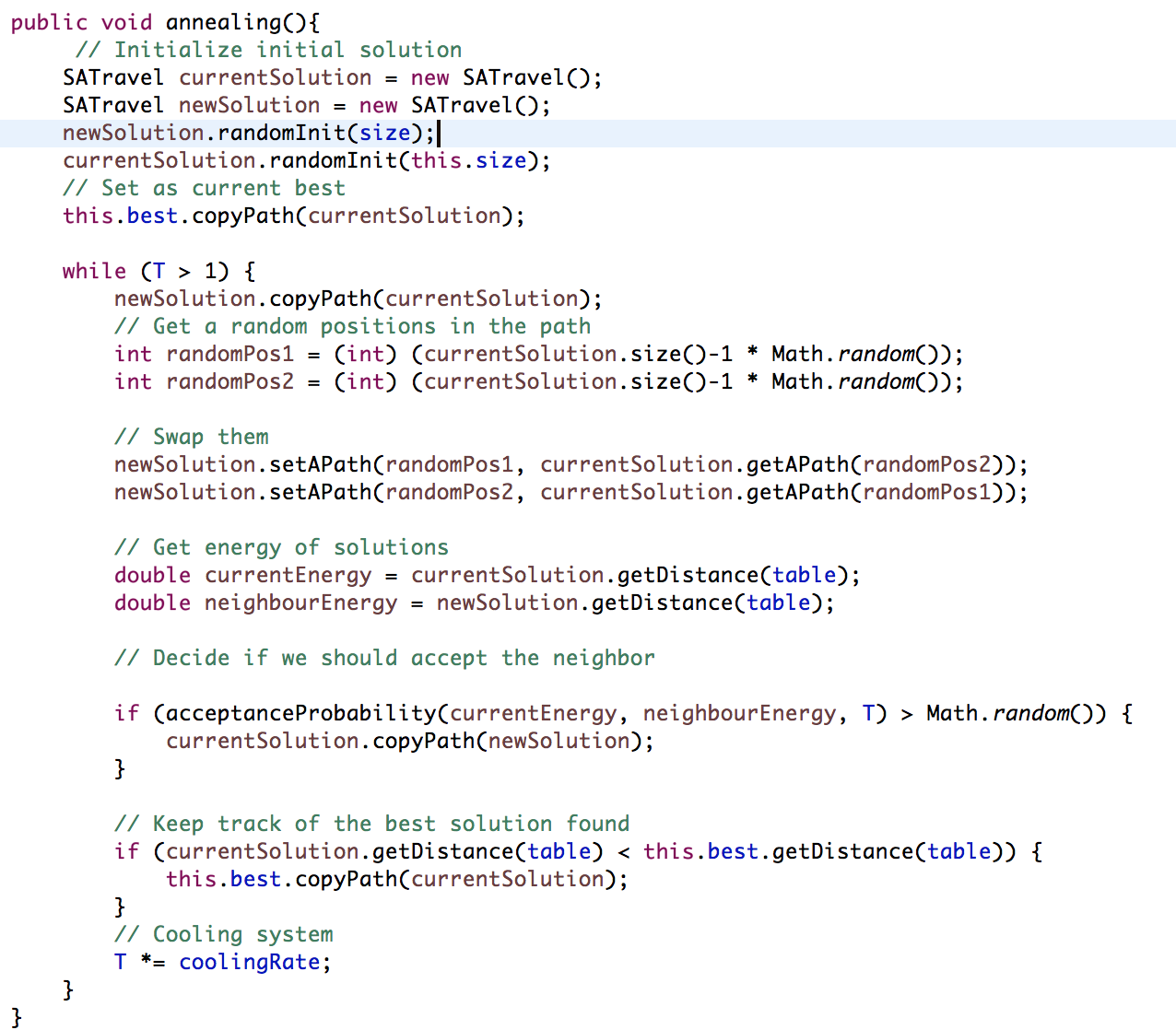
****

**3.2 알고리즘에 대한 세부 구현**

위에서 설명하였듯이 알고리즘은 랜덤 값에서 시작을 하며, 이웃 해를 찾아서 비교하면서 그 과정에서 local optimal을 추적하는 방법으로 동작한다. 여기서 위의 이미지에서 나오는 수용 확률 만큼 결과값에 구애 받지 않고 이웃 해를 따라 가도록 한다. T값이 단계에 따라 줄어들면서 이웃 해의 결과값에 의존적으로 작용하며, 여러 local optimal들 중 best solution을 찾아낸다.



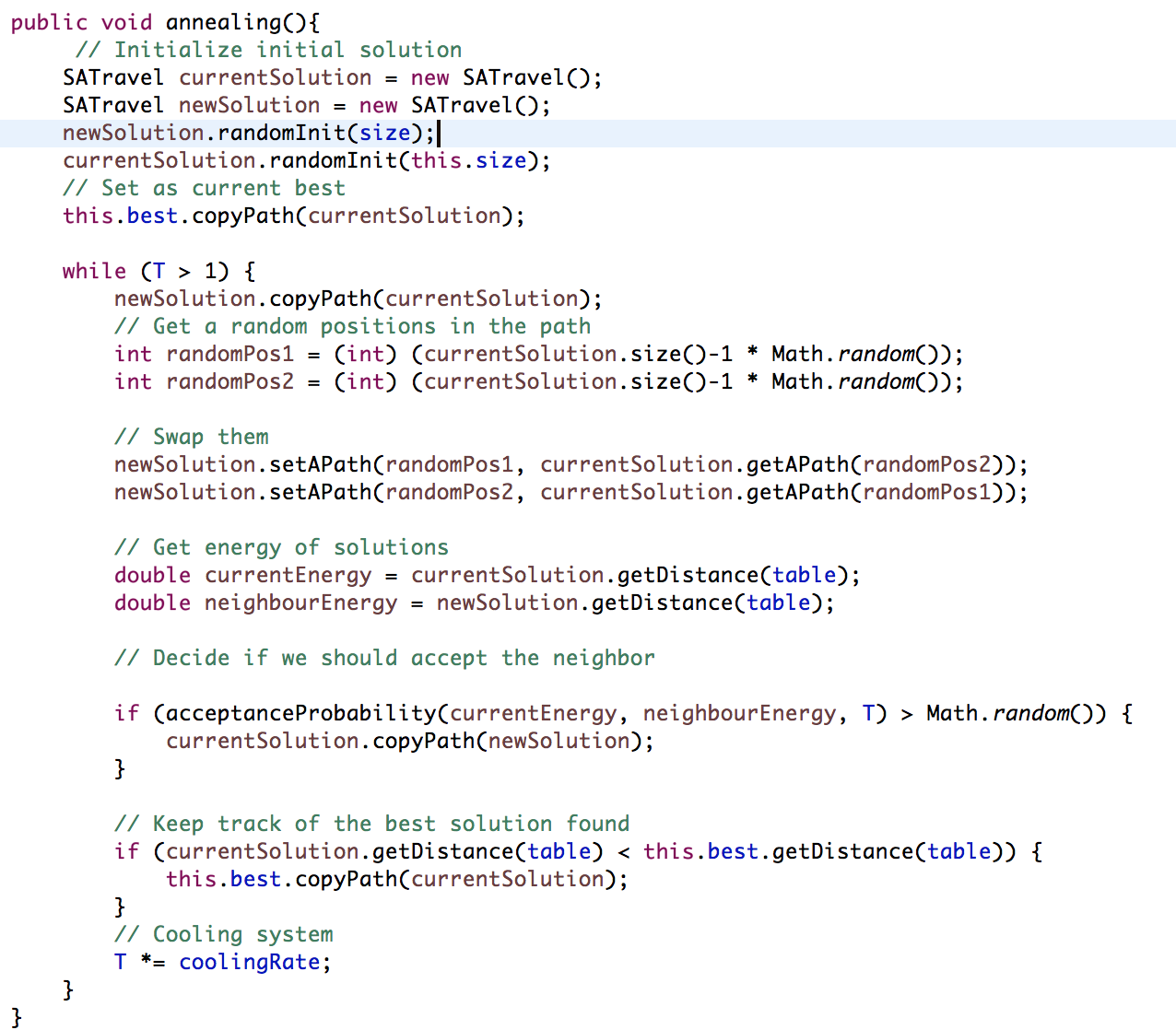
TSP에서 문제의 조건 상 노드 0에서 시작해서 끝나야 하기 때문에 path는 0을 포함하지 않으면서 거리를 계산한 때에만 0에서 출발해서 끝나는 것처럼 구현하였다. Distance의 초기값을 0에서 path의 첫번째 노드로 가는 거리를 주고, path내의 거리를 모두 더한 뒤 마지막 노드에서 다시 0으로 가는 거리를 더해서 distance를 출력한다.



* 메인 어닐링을 수행하는 함수이다. 먼저 path를 randomInit으로 초기화 하고서 best와 currentSolution도 같은 path로 초기화 한다. Newsolution은 currentSolution의 랜덤으로 선택한 두 점을 swap함으로써 결정한다. Newsolution은 currentSolution의 값을 비교해서 더 적절한 값을 선택하는데 이때 수용 확률만큼은 Newsolution을 선택한다.

1. **토의**
   1. **Big-O notation 분석**

시간복잡도를 알기 위해서는 역시 annealing함수를 분석해야한다. 메인이되는 while문은 T에 값에 따라서 줄어들고 cooling rate에의해 줄어드는 정도가 결정된다. 이번 과제에서는 cooling rate가 0.9로 고정되었기 때문에 전적으로 초기 T 값에 의존하게 된다. T값이 로그함수 만큼 감소하며, 한번의 while문 안에서 수행되는 getDistance는 입력사이즈인 n에 의존한다. 결국 annealing과정은 O(n \* logT)가 된다.



하지만 여기서 한가지 더 고려해야한다. 첫 data처리과정에서 2차원 배열의 adjTable을 만드는데에는 n^2만큼의 시간복잡도가 발생하기 때문이다. 따라서 결과는

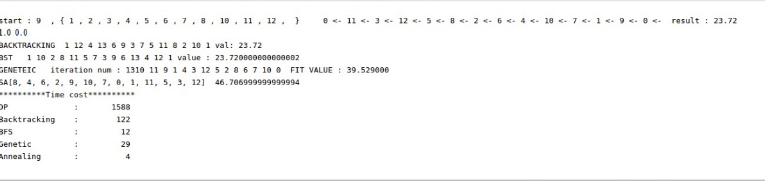
O( max ( logT\*n, n^2) )가 된다.

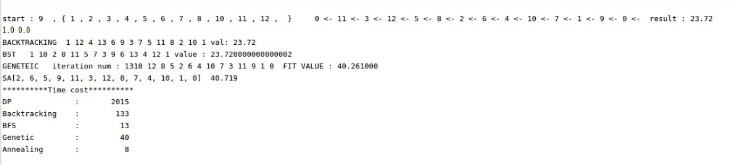
* 1. **구현과정에서 겪은 어려운 점 (시행 착오)**

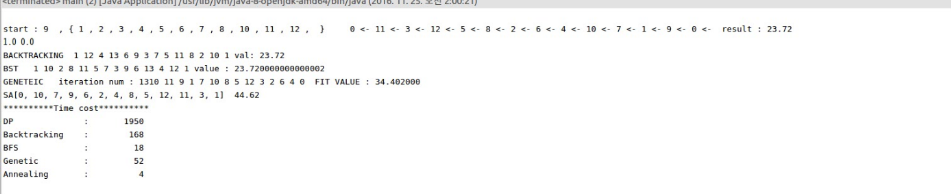
Simulated annealing은 무조건 옵티멀 솔루션을 찾는 알고리즘이 아니기 때문에 수행할 때마다 다른 결과가 나왔다. 이를 최적화 하기 위해서 가장 적절한 결과가 나오도록 T값을 조정하기 위해 여러번 실험을 반복하여야 했다. 정확한 표준이 있는 것이 아니어서 스스로 값을 설정하고 최적화하는 과정이 쉽지 않았다.

**최종 결과 분석**

1. n = 13

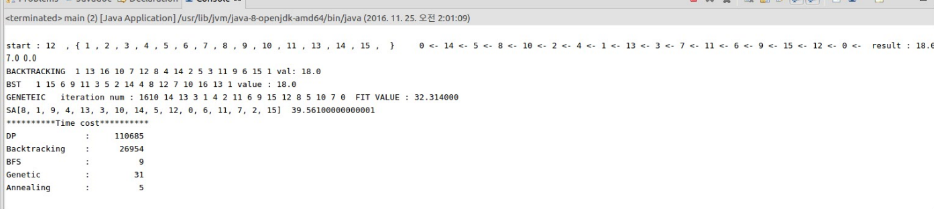






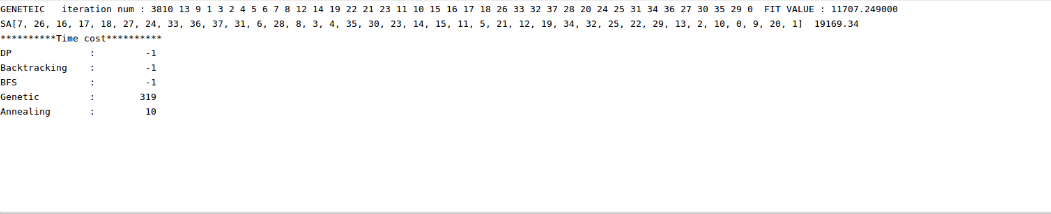
* DP, Backtracking, BST의 결과는 최적 해인 23.72가 나오고 Genetic과 Annealing은 근사 값이 나오는 것을 알 수 있다.
* Genetic과 Annealing 값은 매 수행할 때마다 TSP 값이 계속 다르게 나왔고, 값이 좋을 때와 안 좋을 때의 차이가 있었다.
* DP 의 값이 유달리 크게 나왔는데, 구현 과정 상에서, Space complexity를 줄여 30 이상의 경우에도 코드를 돌아가게 하려고 코드를 짰으나 time complexity를 너무 손실되서 그런걸로 추측한다.
* 이론적으로는 Genetic이 BFS보다도 빨라야 하는데, 객체 간의 값 복사, 이동이 많았떤 genetic 코드 구현 차이로 인해 Genetic의 시간이 좀 더 크게 나왔다.

1. n = 16



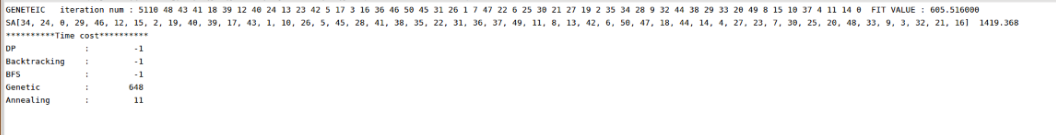
* n은 3밖에 안 늘었지만 에와 backtracking의 값이 엄청나게 증가한 걸 볼 수 있다.
* 반면에 bfs값은 매우 작게 나왔는데, Pruning 과정에서 제거 되는 값들이 적절하게 배치되어 그런 것이라 추측한다.
* Genetic과 annealing값은 전과 큰 차이를 보이지 않았다. 둘 다 적정 값이 되거나 최대 iteration값이 되면 process를 멈추게 했는데 그 과정이 polynomial이서 그런 것이라 추측한다.

1. n = 38

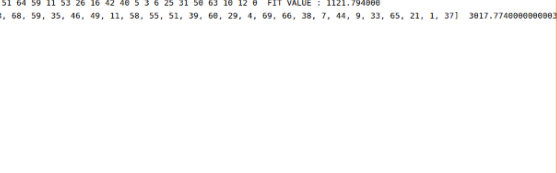
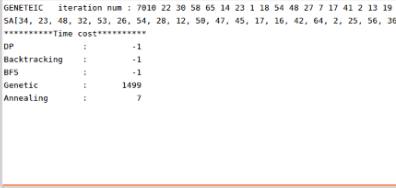


* n = 38인 순간부터 DP, Backtracking, BFS의 값은 너무 실행시간이 오래 걸려서 측정하지 못하였다.
* 반면 Genetic, Annealing은 생각보다 빠른 시간 안에 수행이 되었다. Genetic이 좀 더 오래 걸리는 대신 결과값은 더 좋게 나옴을 확인할 수 있다.

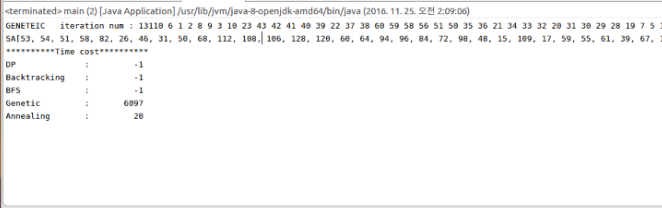
1. n = 51



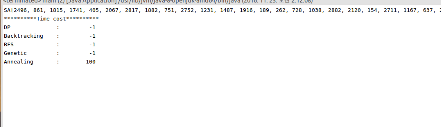
* n= 38과 동일한 결과를 볼 수 있다.
* 시간적으로는 Annealing이 훨씬 빠르지만 optical한 면에서는 Genetic이 더 좋다.
* n = 71

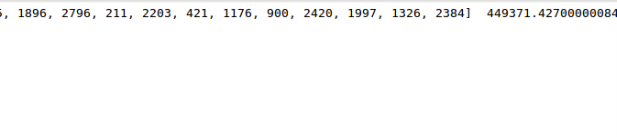


* path의 길이가 너무 길어, 중간 부분은 잘라서 첨부했다.
* n = 131

* n= 2924





* 이 때부턴 Genetic도 시간이 너무 오래 걸려 측정을 하지 못하였다.
* n = 10131
* 
* 위와 같은 오류가 뜨며 input을 받는 것부터 시간이 오래 걸려 측정이 불가능 했다.

정리

* NP문제인 TSP를 해결하는 방식으로, 정확한 값을 구하는 DP, Backtracking, BST와 근사 값을 구하는 Genetic, Annealing에 대해 알아보았다.
* NP solution인 DP, Backtracking, BST는 숫자가 30만 넘어가도 실행이 너무 오래 걸리는 것을 알 수 있었고, 근사 알고리즘은 Genetic과 Annealing은 Polynomial이어서 Input숫자에 그리 큰 영향을 받지 않는 것을 알 수 있다.