UNIT 6: SHORTEST PATH

L15. SINGLE-SOURCE SHORTEST PATHS PROBLEM

- Weighted Graphs
- weights on edges
- General Approach (no particular algorithm)
- Optimal Substructure
- to get efficient complexity

MOTIVATION

Shortest way to drive from A to B Google maps "get directions"

- Formulation: Problem on a weighted graph $G(V, E) \quad W: E \rightarrow \Re$
- Two algorithms: Dijkstra $O(V \lg V+E)$ assumes non-negative edge weights
- Bellman Ford $O(V E)$ is a general algorithm $E=O(V^2)$

- Model as a weighted graph $G(V, E), W: E \rightarrow \Re$
  $-V=$ vertices (street intersections)

$-E$ = edges (street, roads); directed edges (one way roads) $-W(U,V)$ = weight of edge from u to v (distance, toll)

$$\begin{aligned} &\text{path } p = \langle v_0, v_1, \ldots v_k \rangle \\ &(v_i, v_{i+1}) \in E \text{ for } 0 \leq i < k \\ &w(p) = \textstyle\sum_{i=0}^{k-1} w\left(v_i, v_{i+1}\right) \end{aligned}$$

WEIGHTED GRAPHS

Notation:

$v_0 \xrightarrow{p} v_k$ means p is a path from v_0 to v_k. (v_0) is a path from v_0 to v_0 of weight 0.

Definition:

Shortest path weight from u to v as

$$\delta(u,v) = \begin{cases} \min(w(p) : v_0 \xrightarrow{p} v_k) & \text{if } \exists \text{ any such path} \\ \infty & \text{otherwise (unreachable)} \end{cases}$$

Single Source Shortest Paths:

Given $G = (V,E), w$ and a source vertex S, find $\delta(S,V)$ (and the best path) from S to each $v \in V$

Data structures:

$$\begin{aligned} d[v] = &\text{ value inside circle} \\ &= \left\{ \begin{matrix} 0 & \text{if } v = s \\ \infty & \text{otherwise} \end{matrix} \right\} \Longleftarrow \text{ initially} \\ &= \delta(s,v) \Longleftarrow \quad \text{at end} \\ d[v] \geq &\ \delta(s,v) \quad \text{at all times} \end{aligned}$$

$d[v]$ decreases as we find better paths to v

$\Pi[v]$ = predecessor on best path to $v, \Pi[s] = \mathrm{NIL}$

NEGATIVE-WEIGHT EDGES

- Natural in some applications (e.g., logarithms used for weights)
- Some algorithms disallow negative weight edges (e.g., Dijkstra)
- If you have negative weight edges, you might also have **negative weight cycles**
  $\Longrightarrow$ may make certain shortest paths undefined!
- $\Longrightarrow$ If negative weight edges are present, s.p. algorithm should find negative weight cycles (e.g., Bellman Ford)

GENERAL STRUCTURE OF S.P. ALGORITHMS (NO NEGATIVE CYCLES)

Initialize

```
for $v \in V$ : $d[v] \leftarrow \infty$
                $\Pi[v] \leftarrow \mathrm{NIL}$
$d[S] \leftarrow 0$
```

Main

```
repeat
    select edge $(u, v)$    (somehow)
                           [ if $d[v] > d[u] + w(u, v)$ :
    "relax" edge $(u, v)$  [     $d[v] \leftarrow d[u] + w(u, v)$
                           [     $\Pi[v] \leftarrow u$
until all edges have $d[v] \leq d[u] + w(u, v)$
```

OPTIMAL SUBSTRUCTURE

Theorem: Subpaths of shortest paths are shortest paths

Triangle Inequality

Theorem: For all $u, v, x \in X$, we have

$$\delta(u,v) \leq \delta(u,x) + \delta(x,v)$$

THINKING

本讲开始介绍weighted graph（加权图），由此来解决shortest path（最短路径）问题。

同样是从数据结构和算法复杂度的层面来进行分析，某种程度上使用复杂的语言讲解了简单的问题：

- 需要记录节点当前path的长度用于比较：$d[v]$
- 最终节点的最短路径：$\delta[v]$
- 父节点作为树的指针：$\Pi[v]$

但语言的复杂性可能来源于从底层开始建构问题的顺序，也因此引出了很多S.P.中的问题：

1. 权重是负数的话可能会引起负数环，会使得一些节点的最短路径无法定义
2. 在更新节点路径长度的过程中，选择“relax”（松弛）边的方式/顺序

R15. SHORTEST PATHS

BFS FOR SHORTEST PATHS

- split edges (add nodes to them) according to their weights ($<= W$)
- so that the graph turns into an unweighted graph
- use BFS
- $V' = O(WE + V), E' = O(WE) \Longrightarrow$ Complexity: $O(V + WE)$

HOW I MET YOUR MIDTERM

Ted and Marshall are taking a roadtrip from Somerville to Vancouver (that's in Canada). Because it's a 52-hour drive, Ted and Marshall decide to switch off driving at each rest stop they visit; however, because Ted has a better sense of direction than Marshall, he should be driving both when they depart and when they arrive (to navigate the city streets).

- double vertices, each with state odd or even
- edge from odd vertex to even vertex or vice versa
- use Dijkstra's algorithm to find the shortest path
- $V' = 2V, E' = 2E \Longrightarrow$ Complexity: $O(V \log V + E)$

PROBLEM SOLVING IN PRACTICE

Raw Input $\overset{Transform}{\Longrightarrow}$ Input $\Longrightarrow$ **KNOWN ALGORITHMS**
$\Longrightarrow$ Output $\overset{Interpret}{\Longrightarrow}$ Expected Output

THINKING

这节复习课介绍了如何用BFS解决最短路径问题，或者用Dijkstra's Algorithm解决更为复杂的问题，并且介绍了其中的核心思想：**从抽象算法到实际问题的解决思路（见以上草图，重要！调整输入的数据以适应算法或数据结构！）**

L16. DIJKSTRA

DAGS

can have negative values but can't have cycles

Can't have negative cycles because there are **no cycles**!

1. Topologically sort the DAG. Path from u to v implies that u is before v in the linear ordering.
2. One pass over vertices in topologically sorted order relaxing each edge that leaves each vertex.

$\Theta(V+E)$ time

DIJKSTRA'S ALGORITHM

can have cycles but can't have negative values

```
Dijkstra (G, W, s) //uses priority queue Q
  Initialize (G, s)
  S ← None
  Q ← V[G] //Insert into Q
  while Q != None
    do u ← EXTRACT-MIN(Q) //deletes u from Q
    S = S U {u}
    for each vertex v in Adj[u]
      do RELAX (u, v, w) // ← this is an implicit DECREASE KEY
operation
```

- Strategy: Dijkstra is a greedy algorithm: choose closest vertex in $V-S$ to add to set S.
- Correctness: We know relaxation is safe. The key observation is that each time a vertex u is added to set S, we have $d[u]=\delta(s,u)$.

Dijkstra Complexity

$\Theta(v)$ inserts into priority queue

$\Theta(v)$ EXTRACT_MIN operations

$\Theta(E)$ DECREASE_KEY operations

Array impl:

$\Theta(v)$ time for extract min

$\Theta(1)$ for decrease key

Total: $\Theta(V \cdot V + E \cdot 1) = \Theta\left(V^2+E\right) = \Theta\left(V^2\right)$

(where $E=O(V^2)$)

Binary min-heap:

$\Theta(\lg V)$ for extract min

$\Theta(\lg V)$ for decrease key

Total: $\Theta(V \lg V + E \lg V)$

Fibonacci heap (not covered in $6.006)$:

$\Theta(\lg V)$ for extract min

$\Theta(1)$ for decrease key amortized cost

Total: $\Theta(V \lg V + E)$

THINKING

本讲主要介绍了两个针对特殊情况的最短路径算法：

1. 针对没有环的DAG，先进行拓扑排序，然后根据排序结果逐步进行relax即可
2. 针对没有负数的graph，进行类似Uniform-Cost Search的Dijkstra`s Algorithm
    - 添加子路径至Priority Queue，将最小路径及其对应的节点添加至结果，将该节点的子节点对应的路径添加至Priority Queue……（recursion）

对于Dijkstra算法，实际上是把6.001和6.009中的知识系统化了（不过是针对所有节点的最短路径，不是单一目标），可以分析其有效性及复杂度

- 例如需要针对Priority Queue进行Extract_Min以及Insert操作，不同的数据结构相应操作的复杂度不同，导致整个算法的复杂度的不同

R16. RUBIK'S CUBE, STARCRAFT ZERO

THINKING

这节复习课介绍了用最短路径算法解决魔方和简化版星际争霸问题。

从中可以看到很多使用编码解决问题的思路和技巧，尤其是如何将问题本身编码。

想起了6.001中介绍的State Machine，当前State是什么，可能的Process是什么，经过一个Process之后的State是什么，目标State是什么。

所有这一切如何用代码、数据结构和算法表示。

L17. BELLMAN-FORD

generic algorithm (can handle negative cycle)

```
Bellman-Ford(G,W,s)
  Initialize ()
  for i = 1 to |V| - 1
    for each edge (u, v) ∈ E:
      Relax(u, v)
  for each edge (u, v) ∈ E
    do if d[v] > d[u] + w(u, v)
      then report a negative-weight cycle exists
```

THINKING

是一个greedy algorithm

L18. SPEEDING UP DIJKSTRA

DIJKSTRA SINGLE-SOURCE, SINGLE-TARGET

```
Initialize()
Q ← V[G]
while Q != None
  do u ← EXTRACT MIN(Q) (stop if u = t!)
  for each vertex v t Adj[u]
    do RELAX(u, v, w)
```

BI-DIRECTIONAL SEARCH

Alternate forward search from s

backward search from t

(follow edges backward)

df(u) distances for forward search

db(u) distances for backward search

- Termination: Algorithm terminates when some vertex w has been processed, i.e., deleted from the queue of both searches, Q_f and Q_b
- Subtlety: After search terminates, find node x with minimum value of df (x) + db(x). x may not be the vertex w that caused termination as in example to the left!
- Minimum value for df (x) +db(x) over all vertices that have been processed in at least one search

GOAL-DIRECTED SEARCH OR A*

Modify edge weights with potential function over vertices.

$$\bar{w}(u,v) = w(u,v) - \lambda(u) + \lambda(v)$$

THINKING

本讲介绍了Dijkstra的几种变体：单目标（即UCS），双向搜索，A*。

其中双向搜索有一个要点是，奇数路径需要双向搜索交叉后才会有共同的删除的节点，交叉后的半路径的值如果比另一条偶数路径的半路径的值要大，则偶数路径会首先结束搜索。（可以通过算法进行实验，是否可以在有共同的删除的节点后再多走一步，看是否有另一条路径，然后比较两条路径的长短，对于Dijkstra，这个算法应该是可以被证明的）

R18. QUIZ 2 REVIEW

METHODOLOGY

So it's the old problem that we're going through
again and again, **where you have a graph that's 2D,
and we want to compute something that Dijkstra
can't compute on it's own.
Or that Bellman-Ford can't compute on it's own.**

So in order to be able to compute those things,
we need **to add additional states to the graph.
And the way we do that is we make copies of the graph
that we call layers.**

Because we're thinking that if you take that 2D map, and you
create copies of it, you basically
have a 3D graph where there is the original graph.

That's the science part of the problem, the art part of solving the problem is **<u>figuring out what those layers are and how you connect them.</u>** Because by doing that you can solve a ton of problems, as we have seen in this class.

BFS, DFS

BFS	**DFS**
Shortest paths (# paths)	Topological sort
Levels	Edge types: tree, forward, back, cross
	Exit time
Tree: Π parent pointer	Tree: Π parent pointer

THINKING

这节复习课最大的收获是通过实例引出了概括性的内容：如何通过图及其相关算法解决复杂问题——建立层对应节点的不同状态（states）。

<u>PSET 6</u>

6-1

```
StrongestConnection(s, k):
  strength = {s: 0}
  parent = {s: None}
  children = {s: []}
  frontier = [s]
  i = 0
  while frontier and i < k:
    next = []
    for F in frontier:
      for f in F.friends:
        if f is not parent[F]:
          current_strength = strength[F] + ER(F, f)
          if f not in strength:
            strength[f] = current_strength
            parent[f] = F
            children[F].append(f)
            next.append(f)
          else:
            if current_strength > strength[f]:
              strength[f] = current_strength
              parent[f] = F
              UpdateChildrenStrength(f)
    frontier = next
    i = i + 1
```

6-2

(a)

The problem is a dependency graph with no cycle, e.g. a DAG. Therefore we can use topological sort(DFS) to generate the installation order which is $O(V + E)$ time

(b)

```
RemainedInstallationOrder(Rep, WSL, WSL_Dep):
  order = []
  visited = {WSL}
  DFS(Rep, WSL, WSL_Dep, order, visited)
  return order

DFS(Rep, s, Adj, order, visited)
  for v in Adj[s]:
    if v not in Rep and v not in visited:
      DFS(Rep, v, Adj, order, visited)
    order.append[v]
```

Still a DAG, using topological sort, but didn't visit installed libraries and their dependencies(can be proved installed).

Only visit P vertices, thus no more than PD edges. Therefore, $O(P + PD)$ time

THINKING

魔方问题的处理过程中有两个心得：

1. 自上而下的问题解决思路：先写整体问题解决的代码，需要用的的helper function作为占位符写出，然后再一点点填充内容（将复杂问题简单化，乃至可以先写大框架的pseudo code)
   - 自下而上的方法有一个可能的问题是，不确定最底层函数的输入和输出是什么
2. DRY：需要针对两个输入的相同操作交替进行，且一个操作可能会使用到另一个操作的输入时，可以使用别名来进行操作，而不需要写两次代码
   - 例如对于双向的BFS，每次操作重新给别名frontier、parent、other_parent赋值，然后调用helper function即可，从而不需要写两遍几乎相同的代码
   - 在子函数内对于可变数据类型（mutable）的输入变量本身的操作可以帮助解决多次函数操作过程中对于数据的交叉使用问题

    - 注意对于list，0位置的插入和删除操作是线性的，如需要可以换用deque

使用Dijstra解决地图问题时的心得：

1. 在Dijkstra或UCS中，visited和expanded是不同的，一个节点被visited了并不代表下次遇到的时候就不必考虑，因为有不同的路径，但expanded代表该节点的最短路径已经被考虑
2. 最初的方法将路径全部记录下来：由于当前路径不能被修改，因此需要创建新的路径，而创建新路径的复杂度是线性的，创建新路径在循环当中，就给整个算法增加了一个线性因子
    - 6.009的解决方式，使用tuple创建嵌套路径，子路径并没有被修改，新路径只有一个新的节点和指针，创建它是常数的。最后再将嵌套路径展开即可
    - 6.006中的方式是一样的，设置指针，最后通过指针来构建路径
3. 前述的两种方法时间上差异极大，经过多次试验后发现问题可能是：
    - 本lab的priority queue建立了一个key到index的字典，因此层层嵌套的tuple在hash function中的速度极慢
    - 因此如果使用自定义类的一个属性来存储层层嵌套的路径则没有此问题（对该类的hash与其属性无关？）
    - 具体实验过程可以参看Question中的记录
4. **对于抽象的数据结构或算法，面对特定问题不能解决时，不一定是修改抽象数据结构或抽象的算法本身，可以首先尝试修改输入数据的结构，再对算法输出的数据进行解读**：
    - 本例中，为了使数据在Priority Queue中可比较，且保留路径或节点，建立首元素可比较的tuple，或是建立自定义类，定义其比较方式
    - Recitation中介绍的问题，一个节点有多个condition时，建立多layer，并建立各layer之间的边，从而使用已知的针对图的算法

QUIZ 2

4-a

$$f'(x) = 4x^3$$

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{x_i^4 - A}{4x_i^3}$$

4-b

$C = B^3 = A^{3/4} - 6\alpha A^{1/2} + 3\alpha^2 A^{1/4} - \alpha^3$

second and third items could be larger than 1

4-c

compute if $C^4 \le A^3$

4-d

4-e

compute A^3 first, and then use Newton's Method to compute $(A^3)^{1/4}$