

Resum Algorismica Avançada

November 25, 2019

Contents

1	Introducció	1
1.1	Notació Asimptòtica	1
2	Grafs	2
2.1	Estructura de Graf	2
2.2	Definicions	3
2.3	Algoritmes sobre Grafs	3
2.3.1	DFS	3
2.3.2	BFS	4
2.3.3	Dijkstra	5
2.3.4	Bellman-Frod	6
2.4	Flux maxim (s'ha de millorar xd)	7
2.4.1	Ford-Fulkerson	7
3	Gready	8
3.1	Kruskal	8
3.2	Prim	9
4	Programació dinàmica	10
5	Complexitat	11

Resum de teoria de l'assignatura d'informàtica "Algorismica Avançada", basat en les diapositives de teoria penjades al campus.

1 Introducció

1.1 Notació Asimptòtica

- $O(n) \leq n$
- $\Theta(n) = n$
- $o(n) < n$
- $\Omega(n) \geq n$

2 Grafs

Definició 1 (Graf no dirigit). Un graf no dirigit és una parella ordenada $G=(V,E)$, on:

- V conjunt de vertex
- $E \subseteq \{\{x,y\} \mid x,y \in V \wedge x \neq y\}$ conjunt d'arestes

Definició 2 (Graf dirigit). És igual que un graf no dirigit, però els elements de E són parelles ordenades, *i.e.* $E \subseteq \{(x,y) \mid x,y \in V \wedge x \neq y\}$.

Definició 3 (Graf amb pes). Un graf amb pes és un graf on a cada element de E li assignem un valor, que pot representar una distància o alguna altra cosa.

Observació 1. També podem assignar altres qualitats, com color o text, a les arestes, que no cal que tinguin valor numèric.

2.1 Estructura de Graf

Un graf el podem representar de diverses maneres:

1. Donant una **llista linkada** dels seus elements, és a dir donant V i E
2. Donant una matriu amb les "relacions" entre els vertex del graf, a aquesta matriu li diem **Matriu d'adjacència**
3. Donant una **llista d'adjacència**, donar una llista de V on a cada vertex es diu amb quins altres està connectat i com ho està amb cada un d'ells.

Observació 2. Si el graf té pocs vertex, hi ha poques "relacions" entre els diferents vertex, aleshores és poc convenient usar la representació matricial.

Observació 3. Si tots els vertex del graf estan continguts en almenys una aresta, aleshores, a l., per donar tots els seus elements és suficient donar només E , les arestes.

2.2 Definicions

Definició 4 (Grau d'un vertex). El grau d'un vertex és el nombre de vertex al que esta conectat aquest.

Definició 5 (Component conexas). Una component conexas V_1 és un subconjunt de vertex que estan connectats entre si per arestes.

Això és que $V_1 \subseteq V$ i $\forall x, y \in V_1 \exists a_0, \dots, a_n \in E$, amb $a_i = \{z_i, z_{i+1}\}$, on $\forall z_i \in V$ i $z_0 = x, z_n + 1 = y$.

Definició 6 (Cami). A el conjunt ordenat $\{a_0, \dots, a_n\} \subseteq E$ de la definició anterior l'anomenem camí entre x i y .

Definició 7 (Cicle). Un cicle es un camí on $a_0 = a_{n+1}$. Es diu que un graf és acíclic si no conte cicles.

Definició 8 (Graf conex). Diem que un graf és conex quan esta format per una única component conexas.

Definició 9 (Cami d'Euler). És un camí que passa per totes les arestes del graf només una vegada.

Definició 10 (Cicle d'Euler). És un camí d'Euler que comença i acaba en el mateix vertex. Si un graf conte un cicle d'Euler s'anomena Eulerian graph.

Definició 11 (Cami Hamiltonia). És un camí que passa per tots els vertex del graf una vegada.

Definició 12 (Cicle Hamiltonia). És un camí Hamiltonia que comen'a i acaba en el mateix vertex. Un graf que conte un cicle Hamiltonia s'anomena graf Hamiltonia.

2.3 Algoritmes sobre Grafs

2.3.1 DFS

DFS, Depth-First Search, és un algoritme per buscar els vertex accessibles

Pseudo-codi per DFS recursiu:

```
1 function DFS(G, v):  
2     label v as discovered  
3     for all edges from v to w that are in G.adjacentEdges(v) do  
4         if vertex w is not labeled as discovered then  
5             recursively call DFS(G, w)
```

Pseudo-codi per DFS no-recursiu:

```
1 function DFS-iterative(G,v):
2   let S be a stack
3   S.push(v)
4   while S is not empty
5     v = S.pop()
6     if v is not labeled as discovered:
7       label v as discovered
8       for all edges from v to w in G.adjacentEdges(v) do
9         S.push(w)
```

Complexitat (en funció de l'estructura de graf usada):

- Llista linkada = $\Theta(|E|^2)$
- Matriu adjacència = $\Theta(|V|^2)$
- Llista d'adjacència = $\Theta(|E|)$

2.3.2 [BFS](#)

BFS, Breath-First Search, és un algoritme per buscar vertex accessibles. Al contrari que DFS, aquest es basa en amplada, es a dir, explora tots els vertex d'una "profunditat" abans de pasar a la següent. Gracies a això ens permet trobar el camí mínim entre dos vertex.

Pseudo-codi:

```
1 function BFS(G,start_v):
2   let Q be a queue
3   label start_v as discovered
4   Q.enqueue(start_v)
5   while Q is not empty
6     v = Q.dequeue()
7     if v is the goal:
8       return v
9     for all edges from v to w in G.adjacentEdges(v) do
10      if w is not labeled as discovered:
11        label w as discovered
12        w.parent = v
13        Q.enqueue(w)
```

Complexitat $O(|V| + |E|)$, si es coneix el nombre de vertex del graf i s'usa una estructura adequada la complexitat es pot reduir a $O(|V|)$.

2.3.3 Dijkstra

Dijkstra és un algoritme per a trobar camins mínims entre vertex. La diferencia entre Dijkstra i BFS és que Dijkstra té en conte el pes de cada aresta, mentre que pel contrari BFS no.

Pseudo-codi:

```

1 function Dijkstra(Graph, source):
2
3     create vertex set Q
4
5     for each vertex v in Graph:
6         dist[v] = INFINITY
7         prev[v] = UNDEFINED
8         add v to Q
9     dist[source] = 0
10
11     while Q is not empty:
12         u = vertex in Q with min dist[u]
13
14         remove u from Q
15
16         for each neighbor v of u: // only v that are still in Q
17             alt = dist[u] + length(u, v)
18             if alt < dist[v]:
19                 dist[v] = alt
20                 prev[v] = u
21
22     return dist[], prev[]

```

Observacio 4. Si només busquem el camí mínim fins a un cert vertex, aleshores després de la línia 14 si u és l'objectiu ja podem parar l'algorisme i fer el "backtracking".

Observacio 5. Si en ves de tenir un conjunt Q tenim una cua de preferència, aleshores a la línia 12 quan haguem de trobar l'element mínim no haurem de fer res perquè ja estarà al principi de la cua. Però llavors haurem de modificar la prioritat de la cua en cada iteració.

Complexitat

$$O(|E| \cdot T_{dk} + |V| \cdot T_{em}),$$

on T_{dk} i T_{em} són les complexitats de *decrease – key* i de *extract – minimum* respectivament.

- Usant una llista linkada = $O(|V|^2)$.
- Usant una cua de prioritats, [Binary Heap](#), = $O((|V| + |E|)\log(|V|))$.

Observacio 6. Dijkstra troba el camí mínim si tots els pesos són no negatius.

2.3.4 Bellman-Ford

Bellman-Ford és un algorisme equivalent a Dijkstra que tot i ser menys eficient és més versàtil ja que també funciona amb pesos negatius.

Pseudo-codi

```
1 function BellmanFord(list vertices, list edges, vertex source)
2   :: distance[], predecessor[]
3
4   // This implementation takes in a graph, represented as
5   // lists of vertices and edges, and fills two arrays
6   // (distance and predecessor) about the shortest path
7   // from the source to each vertex
8
9   // Step 1: initialize graph
10  for each vertex v in vertices:
11    distance[v] := inf      // Initialize the distance to all
12    vertices to infinity
13    predecessor[v] := null  // And having a null predecessor
14
15  distance[source] := 0      // The distance from the source to
16  itself is, of course, zero
17
18  // Step 2: relax edges repeatedly
19  for i from 1 to size(vertices)-1:
20    for each edge (u, v) with weight w in edges:
21      if distance[u] + w < distance[v]:
22        distance[v] := distance[u] + w
23        predecessor[v] := u
24
25  // Step 3: check for negative-weight cycles
26  for each edge (u, v) with weight w in edges:
27    if distance[u] + w < distance[v]:
28      error "Graph contains a negative-weight cycle"
29
30  return distance[], predecessor[]
```

Observacio 7. Hi ha un problema i és que podem trobar cicles amb cost negatiu.

Complexitat $O(|V| \cdot |E|)$.

2.4 Flux maxim (s'ha de millorar xd)

A teoria de grafs, una xarxa de flux és un graf dirigit que conte un origen, source S , i una desembocadura, sink T , entre altres nodes connectats amb edges. Cada edge te una capacitat maxima de flux que pot assumir. El flux dins la xara ha de complir les següents condicions:

- $\forall e_i \in E$ amb $e_i \neq S$, $e_i \neq T$, es té que el flux que entra i el que surt són iguals, *i.e.* $\sum f_{input} = \sum f_{output}$, equivalentment $\sum_{\{u:(u,v) \in E\}} f_{uv} = \sum_{\{w:(v,w) \in E\}} f_{vw}$.
- $\forall e_i \in E$ es té que $0 \leq flow(e_i) = |f_{e_i}| \leq Capacity(e_i)$.
- El flux total $|f|$ que surt de S és igual al flux total que arriba a T , *i.e.* $|f| = \sum_{\{v:(S,v) \in E\}} f_{Sv} = \sum_{\{v:(v,T) \in E\}} f_{vT}$.
- El flux entre dos vertex u i v , $f(u, v)$ compleix la simetria $f(u, v) = -f(v, u)$.

Definició 13 (Flux maxim). El flux maxim és la maxima quantitat de flux que la xarxa pot suportar que passi desde S fins a T , *i.e.* el maxim $|f|$.

Definició 14 (Tall $s - t$). Un tall $C = (A, B)$, amb $A, B \subset V$ i $A \cup B = V$, és una partició dels nodes, tal que $S \in A$ i $T \in B$, i amés tenim que $X_C = \{(u, v) \in E \mid u \in A, v \in B\}$. I la capacitat del tall és $Capacity(A, B) = \sum_{(u,v) \in X_C} c_{uv}$

Teorema 1 (max-flow min-cut). El maxim flux entre S i T és igual a la minima capacitat d'entre tots els talls que divideixen la xarxa.

Definició 15 (Xarxa Residual). La xarxa residual consisteix en edges que admeten més flux. Sigui $G = (V, E)$ una xarxa de flux amb inici S i destinacio T . Sigui f el seu flux, aleshores donats $u, v \in V$ la quantitat de flux addicional que accepte aquesta aresta és la seva capacitat residual $c_f(u, v)$, *i.e.* $c_f(u, v) = c(u, v) - f(u, v)$.

2.4.1 Ford-Fulkerson

És un algoritme greedy per a trobar el flux maxim d'una xarxa de flux.

Pseudo-codi¹:

```

1 function: FordFulkerson(Graph G, Node S, Node T):
2   Initialise flow in all edges to 0
3   while (there exists an augmenting path(P) between S and T in
   residual network graph):
4     Augment flow between S to T along the path P
5     Update residual network graph
6   return
```

¹<https://www.hackerearth.com/practice/algorithms/graphs/maximum-flow/tutorial/>

3 Greedy

Un algoritme greedy és un algoritme que segueix la heurística solucionar problemes a partir de dividir aquests en sub-problemes i trobar les solucions òptimes locals a aquests per tal de donar una solució global. En molts problemes l'heurística greedy no troba el resultat òptim del problema.

En general els algoritmes greedy consta de les següents parts

- Un conjunt de candidats d'on es treura la solució.
- Una funció de selecció que triara el millor candidat i l'afegira a la solució.
- Una funció que determina la validesa del candidat per estar a la solució.
- Una funció que assigna un valor (heuristic) a les solucions parcials.
- Una funció que ens determini si hem arribat a la solució.

Els algoritmes greedy tenen la propietat que mai reavaluen les sub-solucions que ja han trobat.

Definició 16. Diem que un problema té una subestructura òptima si la seva solució òptima conté les solucions òptimes dels seus sub-problemes.

Observació 8. Els algoritmes greedy són útils per a solucionar problemes que tenen una subestructura òptima.

Definició 17. Un arbre és un graf sense cicles.

Definició 18. Donat un graf G connex qualsevol, el seu Minimum-spanning-tree, MST , és l'arbre que té el menor pes possible contenint tots els nodes de G .

Observació 9. Notem que el problema de trobar el MST d'un graf admet una subestructura òptima.

3.1 Kruskal

És un algoritme que troba el minimum-spanning-tree d'un graf.

Observació 10. Si el graf no és connex aleshores troba el MST per a cada una de les seves components connexes.

Pseudo-codi

```

1 function KRUSKAL(G):
2     A = void
3     foreach v in G.V:
4         MAKE-SET(v)
5     foreach (u, v) in G.E ordered by weight(u, v), increasing:
6         if FIND-SET(u) different of FIND-SET(v):
7             A = A ∪ {(u, v)}
8             UNION(FIND-SET(u), FIND-SET(v))
9     return A

```


on

MAKE-SET(v): Crea un singleton amb v .

FIND-SET(v): Troba el conjunt que conté l'element v .

UNION(B, C): Fa la unió dels conjunts B i C .

Complexitat $O(E \log E)$

3.2 Prim

És una alternativa a Kruskal. Però només funciona amb grafs connexos.

Pseudo-codi:

```
1 function Prim(G,s):
2   for each vertex v in graph G:
3     cost(v) = infinity
4     prev(v) = nil
5     PQinsert(v, Q)
6   cost(s) = 0
7
8   Q = makequeue() #priority queue with cost as key
9   while Q not empty:
10    v = pop_min(Q)
11    for each edge (v,u) | u is in Q:
12      if cost(u) > ueight((v,u))
13        cost(u)=ueight((v,u))
14        prev(u) = v
15        decreasekey(Q,u)
```

Complexitat $O(E \log E)$

4 Programació dinàmica

5 Complexitat

Teorema 2 (Master Theorem). Anàlisi de la complexitat de algoritmos recursivos.

Siendo $T(n)$ la complexitat del algoritmo, $f(n)$ la complexitat del caso base, a el número de subproblemas a cada nivel de recursión y b el factor por el que dividimos la entrada, denotamos la complexitat de un algoritmo recursivo como:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Si notamos la función $f(x)$ en base a su complexitat de la forma $O(n^d)$ obtenemos:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n)$$

Y según el teorema masters podemos afirmar que:

$$\begin{aligned} T(n) &= O(n^d) && \text{if } d > \log_b a \\ T(n) &= O(n^d \log n) && \text{if } d = \log_b a \\ T(n) &= O(n^{\log_b a}) && \text{if } d < \log_b a \end{aligned}$$