Estructuras de Datos

Grafos: búsqueda de caminos.
Caminos mínimos desde un nodo.

ChangeLog

6/5/2025
- Versión adaptada de la versión con cursor.

21/5/2025
- Reordenada secuencia de transparencias.

Contenidos

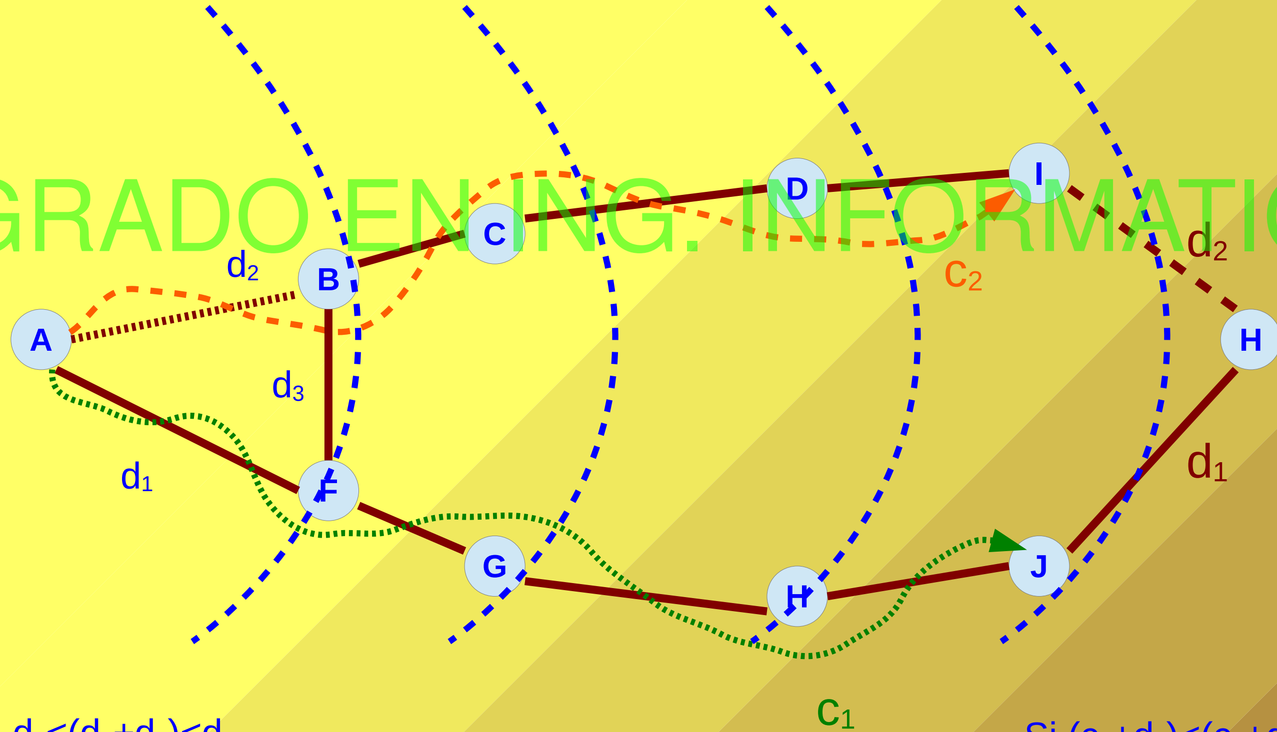
- ¿Están dos vértices conectados?: Algoritmo de Warshall.
- **Todos los caminos mínimos desde un vértice origen al resto: Algoritmo de Dijkstra.**
- **El camino mínimo entre dos vértices: Algoritmo A*.**
- Todos los caminos mínimos entre todos los pares de vértices: Algoritmo de Floyd.

Búsqueda de caminos

- Algoritmo de Dijkstra.
  - **Caminos con distancia mínima** desde un vértice al resto.
  - Se aplica en **grafos ponderados** (El peso del lado se entiende como una distancia >0 (∞ si no hay adyacencia).
  - Es un **recorrido en amplitud** pero utilizando una **cola con prioridad** usando la distancia mínima hasta el momento (HeapMin).
  - **Algoritmo voraz**: en cada iteración se encuentra la mejor solución local: el vértice no visitado con menor distancia al origen (la cabeza de la cola) y no puede después haber otra mejor.
  - Su complejidad es $O(N^2)$.

Búsqueda de caminos

- Algoritmo de Dijkstra: ¿por qué funciona?



Si $d_1<(d_1+d_3)<d_2$
PQueue:[...,(d_1,F,A), (d_1+d_3,B,F), (d_2, B,A), ...]

Si $(c_1+d_1)<(c_2+d_2)$
PQueue:[..., (c_1+d_1,H,J), (c_2+d_2, H,I), ...]

Búsqueda de caminos

- Algoritmo de Dijkstra.

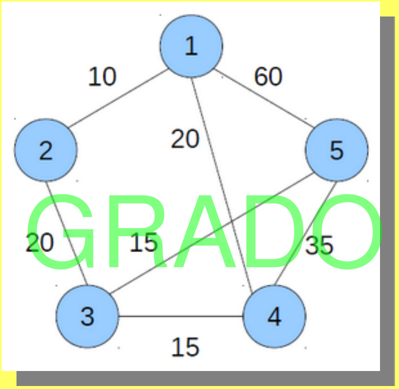
```
Algorithm Dijkstra(Var g:Graph[V, Float], start:Vertex[V];
                   Var P:Array[Int], D:Array[Float]) //O(N²)
Var
  u,v: Vertex[V]
  e: EdgeIterator[V, Float]
  t: Tuple[Float, Int, Int] //Tuple(distance, vertex’s label, predecesor’s label)
  q: PriorityQueue[Tuple[Float, Int, Int], Less]
Begin
  P ← Array[Int]::make(g.nVertices()) //Predecessors.
  D ← Array[Float]::make(g.nVertices()) //Min Distances.
  D ← ∞ //Initialize distances.
  g.reset()
  q.enque(Tuple(0.0, start.label(), start.label()))
  While Not q.isEmpty() Do
    t ← q.front()
    q.deque()
    u ← g.vertex(t[1])
    If Not u.isVisited() Then
      P[t[1]] ← t[2]
      D[t[1]] ← t[0]
      u.setVisited(True)
      e <- g.edgesBegin(u)
      While e <> g.edgesEnd(u) Do
        v ← e.get().other(u)
        If not v.isVisited() Then
           q.enque(Tuple(D[u.label()]+e.get().item(), v.label(), u.label()))
        e.gotoNext()
      End-While
    End-If
End.
```

Se asume que:

- Los vértices tienen etiquetas 0,1, 2,…, N-1
- La cola es un HeapMin.
- Las tuplas se ordenan de forma lexicográfica.

Búsqueda de caminos

- Algoritmo de Dijkstra: ejemplo.



1: {**<0,1,1>**}
2: {**<10,2,1>**, <20,4,1>, <60,5,1>}
3: {**<20,4,1>**, <30,3,2>, <60,5,1>}
4: {**<30,3,2>**, <35,3,4>, <55,5,4>, <60,5,1>}
5: {<35,3,4>, <45,5,3>, <55,5,4>, <60,5,1>}
6: {**<45,5,3>**, <55,5,4>, <60,5,1>}
7: {<55,5,4>, <60,5,1>}
8: {<60,5,1>}

u	P	D
1	1	0
2	-	∞
3	-	∞
4	-	∞
5	-	∞

iter 1

u	P	D
1	1	0
2	1	10
3	-	∞
4	-	∞
5	-	∞

iter 2

u	P	D
1	1	0
2	1	10
3	-	∞
4	1	20
5	-	∞

iter 3

u	P	D
1	1	0
2	1	10
3	2	30
4	1	20
5	-	∞

iter 4

u	P	D
1	1	0
2	1	10
3	2	30
4	1	20
5	3	45

iter 6

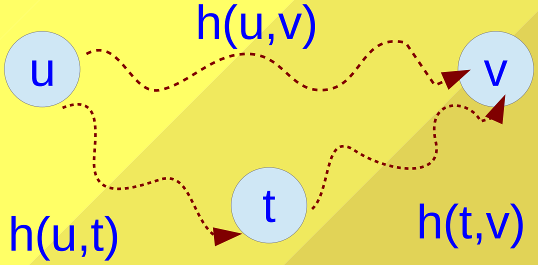
iter 5 “vacía”

¿Cómo recuperar un camino?

Camino para llegar a 5:
P[5] = 3
P[3] = 2
P[2] = 1
P[1] = 1 <stop>
Camino: 1→2→3→5

Búsqueda de caminos

- Algoritmo. A*.
  - Camino mínimo entre dos vértices.
  - Se trata de el algoritmo de Dijkstra modificado $O(N^2)$.
  - **Heurística h(u,v)**: función que devuelve una estimación de la distancia entre dos vértices.
  - La heurística h(u,v) debe cumplir la desigualdad triangular:
    - h(u,v) <= h(u,t)+h(t,v)
  - Cuando se expande un vértice t, la cola se ordena por el campo: distancia[inicio, t]+ h(t,destino)
  - El algoritmo termina cuando se visita el vértice destino o la cola queda vacía (el vértice origen no está conectado con el destino).



H debe cumplir la desigualdad triangular: $h(u,v) \leq h(u,t) + h(t,v)$

Búsqueda de caminos

- Algoritmo de A*.

```
Algorithm A*(Var g:Graph[V, Float], start,end:Vertex[V], h:Heuristic,
             Var P:Array[Int], D:Array[Float]
Var
  u,v: Vertex[V]
  e: EdgeIterator[V, Float]
  t: Tuple[Float, Float, Int, Int]  //(dist+h,dist,vertex’s label, predecessor’s label)
  q: PriorityQueue[Tuple[Float, Float, Int, Int], Less]
Begin
  P ← Array[Integer]::make(g.nVertices())
  D ← Array[Float]::make(g.nVertices())
  D ← ∞
  q.enque(Tuple(h(start, end), 0.0, start.label(), start.label())
  g.reset()
  While Not q.isEmpty() And Not end.isVisited() Do
    t ← q.front()
    q.deque()
    u ← g.vertex(t[2])
    If Not u.isVisited() Then
      P[t[2]] ← t[3]
      D[t[2]] ← t[1]
      u.setVisited(True)
      e ← g.edgesBegin(u)
      While e<>g.edgesEnd(u) Do
        v ← e.get().other(u)
        If Not v.isVisited() Then
           q.enque(Tuple(D[t[2]]+e.get().item()+h(v, end),D[t[2]]+e.get().item(),v.label(),t[2])
        e.gotoNext()
      End-While
    End-If
End.
```

Se asume que:
- Los vértices tienen etiquetas 0,1, 2,…,N-1
- La cola es un HeapMin.
- Las tuplas se ordenan de forma lexicográfica.
- H(u,v) <= {long. cualquier camino que conecte u con v}
- Cuando H(u,v)=0 para todo u,v es el algoritmo de Dijsktra

Búsqueda de caminos

- Resumiendo:
  - Si queremos encontrar los caminos con mínima distancia desde un origen.
    - Algoritmo de Dijkstra $O(N^2)$.
  - Si queremos encontrar el camino mínimo entre dos vértices.
    - Algoritmo A* $O(N^2)$.
    - Necesitamos una heurística para estimar el camino que falta para llegar al destino.

Referencias

- Lecturas recomendadas:
  - Caps. 14, 15 y 16 de “Estructuras de Datos”, A. Carmona y otros. U. de Córdoba. 1999.
  - Wikipedia:
    - Alg. Warshall: en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
    - Alg. Dijkstra: en.wikipedia.org/wiki/Dijkstra%27s_algorithm
    - Alg. A*: en.wikipedia.org/wiki/A*_search_algorithm
    - Alg. Floyd:en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

Estructuras de Datos

Grafos: búsqueda de caminos.

Caminos mínimos desde un nodo.