Muy buena Tarea.

Juan Cisneros Resendiz
Algoritmos en gráficas 2021
Tarea 6
Fecha de entrega: 13 de marzo

El objetivo de esta tarea es que entiendas mejor la estructura de los caminos más cortos y por ende las ideas que subyacen en los algoritmos que vimos en clase para resolver tal problema.

1. En este problema vas a explorar tu creatividad para diseñar algoritmos. Sea G una gráfica dirigida con pesos en sus aristas y sean $s, t \in V(G)$, en este problema deberás diseñar una estrategia para encontrar un camino (más) corto entre s y t. Tienes algunas opciones a elegir:

a. Diseñar un algoritmo que calcule un camino más corto entre s y t. Si eliges esta opción deberás diseñar (o al menos intentar diseñar) un algoritmo que reciba como entrada a G y a los dos vértices, y devuelva como salida el camino más corto entre ambos, puedes suponer que la gráfica no tiene pesos negativos en sus aristas. Tu objetivo deberá ser, además de generar un algoritmo correcto, intentar optimizar el tiempo de ejecución de tu algoritmo (analiza su complejidad), intenta demostrar que tu algoritmo calcula correctamente el camino más corto.

b. Diseñar una heurística que calcule un camino corto entre s y t. Si eliges esta opción, deberás intentar optimizar el parámetro de aproximación a la longitud del camino más corto. Intenta demostrar cuál es la cota superior de tu parámetro de aproximación.

c. Diseña un algoritmo que funcione correctamente para alguna familia de gráficas dirigidas. Por ejemplo, en clase vimos que el algoritmo BFS devuelve caminos más cortos si la gráfica de entrada no tiene pesos en las aritas, y que DFS devuelve caminos más cortos si la gráfica de entrada es DAG. En ese mismo sentido, si eliges esta opción, debrás diseñar un algoritmo que reciba como entrada una gráfica que pertenezca a la familia de gráficas que tú decidas, y que devuelva como salida el camino más corto entre s y t. Igual que en el inciso a, deberás intentar optimizar la complejidad de tu algoritmo (analiza la complejidad), intenta demostrar que tu algoritmo calcula correctamente el camino más corto.

Para cualquiera de las tres opciones deberás probar tu algoritmo en algunas gráficas generadas en Sage. Para cada gráfica genera el camino más corto utilizando la función de Sage que lo calcula (usando Dijkstra o Bellman-Ford), compara la longitud de dicho camino con el devuelvo por tu algoritmo. Discute el comportamiento de tu algoritmo o de tu heurística. Incluye un párrafo en tu reporte donde me cuentes las ideas que usaste para generar tu algoritmo.

El algoritmo diseñado funciona sobre gráficas en las cuales todas sus aristas tienen peso uno. La descripción del problema es la siguiente, sobre una gráfica se busca el camino mas corto entre s y t. Podemos buscar esos caminos simplemente usando un recorrido BFS, pero hay casos en los cuales hacer un recorrido BFS es muy tardado, por ejemplo en un árbol binario, buscar un camino desde la raíz s hasta una hoja t toma tiempo $O(2^h)$, donde h es la altura del árbol.

La idea del algoritmo propuesto es la siguiente: en lugar de un solo recorrido BFS que comienza desde el vértice inicial s, ahora también haremos un recorrido BFS que comience desde el vértice final t, y haremos estos dos recorridos "simultáneamente"hasta encontrar el camino más corto de s a t.

Podemos imaginarnos a los dos recorridos BFS como dos círculos, uno con el centro el vértice s, y el otro con el centro el vértice t, los cuales contienen todos (y solamente) los vértices que están a distancia d_s y d_t, respectivamente (Figura 2). En cada "paso"del algoritmo incrementamos el radio de alguno de los círculos, hasta que haya una intersección entre los dos círculos, cualquier vértice de esa intersección forma parte de un camino más corto entre s y t (falta demostrarlo), así que tomamos uno arbitrariamente.

Para poder ejecutar los recorridos BFS de esa manera (como círculos), suponiendo que comenzamos el algoritmo con el BFS de s, tenemos que decidir si hacer ahora el recorrido BFS de t o seguir con el recorrido de s solo cuando se esté en el final de una etapa en el sentido del algoritmo BFSWithToken, que es cuando

se tienen en la cola solo los vértices de distancia i. Más adelante veremos que el orden en que elegimos los recorridos influye en el tiempo de ejecución del algoritmo.

El tiempo de ejecución de los recorridos BFS para el cálculo de los recorridos más cortos se ve dominado por el paso de inicialización InitSSSP(s), que toma tiempo $O(V)$, es por eso que en este algoritmo se utilizará, en lugar de ese paso de inicialización, una tabla hash para guardar las distancias $dist(v)$, la cual tiene como valor por defecto ∞ cuando no se le ha asignado un valor a la distancia del vértice v, y otra tabla hash para $pred(v)$, con valor por defecto Null.

El algoritmo es el siguiente:

```
BFSDoubleEnded(s, t):
dist_s(s) ← 0   «distancia desde el vértice s»
dist_t(t) ← 0   «distancia desde el vértice t»
queue_s.Push(s)   «cola del recorrido BFS de s»
queue_t.Push(t)   «cola del recorrido BFS de t»
queue_s.Push(*)   «cola del recorrido BFS de s»
queue_t.Push(*)   «cola del recorrido BFS de t»
sel ← 's'   «selección del recorrido BFS inicial»
while queue_s and queue_t are not both empty:
  s_sel ← queue_s.Peek()   «mira el vértice siguiente en salir de la cola sin sacarlo»
  t_sel ← queue_t.Peek()
  if s_sel = '*' and sel = 's'   «si ya terminamos una etapa del BFS de s»
    sel ← «estrategia de selección de recorrido»
    queue_s.Pop()
    queue_s.Push(*)
  else if t_sel = '*' and sel = 't'   «si ya terminamos una etapa del BFS de t»
    sel ← «estrategia de selección de recorrido»
    queue_s.Pop()
    queue_s.Push(*)
  if sel = 's'   «elegimos el recorrido de s»
    u ← queue_s.Pop()
    for all edges u → v
      if dist_s(v) > dist_s(u) + 1
        dist_s(v) ← dist(u) + 1
        pred_s(v) ← u
      «si el nodo fue visitado desde s y desde t, entonces hay una intersección entre los recorridos BFS»
      if (visitado(v, 's')) and (visitado(v, 't'))
        return v
      else
        queue_s.Push(v)
  if sel = 't'   «elegimos el recorrido de t»
    u ← queue_s.Pop()
    for all edges v → u
      if dist_t(v) > dist_t(u) + 1
        dist_t(v) ← dist_t(u) + 1
        pred_t(v) ← u
      «si el nodo fue visitado desde s y desde t, entonces hay una intersección entre los recorridos BFS»
      if (visitado(v, 's')) and (visitado(v, 't'))
        return v
      else
        queue_t.Push(v)
```

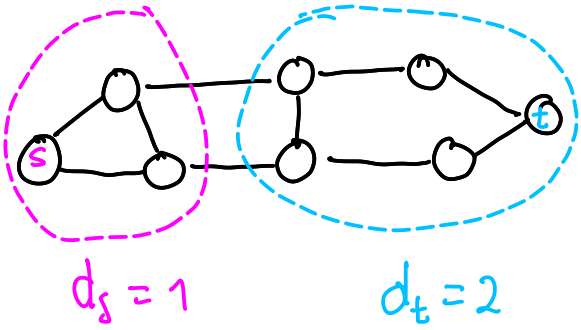


Figura 1: Ejemplo de los círculos del algoritmo

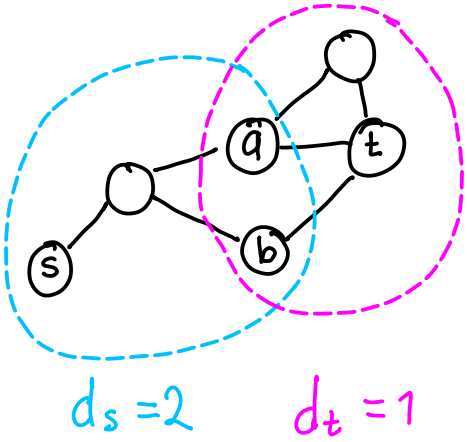
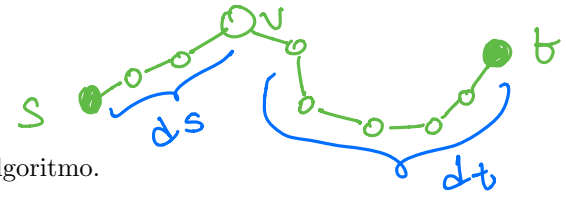


Figura 2: Intersección de los círculos que forman los recorridos BFS de s y de t

¿No debería ser teorema decir: el algoritmo es correcto?



Primero demostremos la correctitud de este algoritmo.

Theorem 1. *Sean d_s y d_t la fase actual de $queue_s$ y $queue_t$, respectivamente, y esas dos colas están en inicio de su fase (todos los vértices v de $queue_s$ tienen distancia d_s desde s hacia v, lo mismo para $queue_t$), tal que es el primer inicio de etapa en el cual hay vértices v que están tanto en $queue_s$ como en $queue_t$, entonces un camino más corto de s a t está compuesto de un camino más corto de s a v, y de un camino más corto desde v a t, donde v es un vértice que está en las dos colas. Ese camino tiene longitud $d_s + d_t$*

Supongamos por contradicción que existe un camino más corto que no pasa por algún vértice que está en las dos colas, sin pérdida de la generalidad recorramos ese camino comenzando en s, ese camino más corto tiene que pasar por algún vértice x tal que $d(s, x) = d_s$. De manera similar, al recorrer ese camino desde t, tiene que pasar por algún vértice y tal que $d(t, y) = d_t$, ya que tanto x como y no están en las dos colas, entonces para conectar a x como a y tiene que haber una arista (x, y), ese camino tiene longitud $d_t + 1 + d_s$, eso contradice que es un camino más corto (Figura).

¿..?

Ahora veremos dos estrategias de selección de recorrido.

La primera es ir cambiando de recorrido del BFS de s al BFS de t cada ves que finaliza una etapa, esa estrategia tiene como característica adicional que encuentra el vértice que está en el "centro" del camino

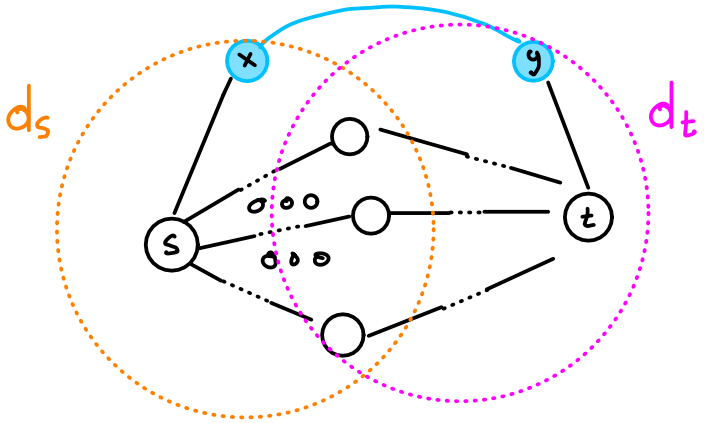


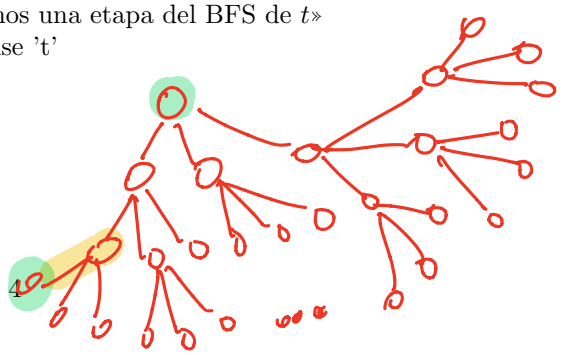
Figura 3: El camino tiene longitud $d_s + d_t + 1$

más corto, en particular si la distancia del camino más corto es par, las dos circunferencias de los dos recorridos BFS tendrán el mismo radio d cuando intersecten. En el caso que la distancia mínima sea impar, una circunferencia tendrá radio $d+1$ y la otra d, así que en ese caso el vértice encontrado estará una unidad más cercano al vértice inicial (o final) que al otro. Esta estrategia se presenta en el algoritmo BFSDoubleEnded1

BFSDoubleEnded1(s, t):
  ...
  if s_{sel} = '*' and sel = 's' «si ya terminamos una etapa del BFS de s»
    $sel \leftarrow$ 't'
    $queue_s$.Pop()
    $queue_s$.Push(*)
  else if t_{sel} = '*' and sel = 't' «si ya terminamos una etapa del BFS de t»
    $sel \leftarrow$'s'
    $queue_t$.Pop()
    $queue_t$.Push(*)
  ...

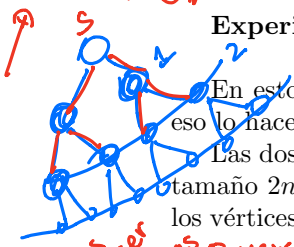
La segunda estrategia es tomar en cada etapa la cola que tenga menos elementos. Al parecer el mejor caso para esta estrategia es la búsqueda en un árbol q-ario, donde el vértice inicial es la raíz y el vértice final una de sus hojas. En la búsqueda con del camino más corto con el BFS ordinario, si empezamos con el vértice raíz esta búsqueda toma tiempo $O(n)$, pero con esta estrategia el algoritmo BFSDoubleEnded escogerá siempre expandir la hoja, así que este algoritmo encuentra el camino más corto en $O(\frac{n}{q})$ en ese caso.

BFSDoubleEnded2(s, t):
  ...
  if s_{sel} = '*' and sel = 's' «si ya terminamos una etapa del BFS de s»
    $sel \leftarrow$'s' if $queue_s$.Size() $\leq$ $queue_t$.Size() else 't'
    $queue_s$.Pop()
    $queue_s$.Push(*)
  else if t_{sel} = '*' and sel = 't' «si ya terminamos una etapa del BFS de t»
    $sel \leftarrow$'s' if $queue_s$.Size() $\leq$ $queue_t$.Size() else 't'
    $queue_t$.Pop()
    $queue_t$.Push(*)
  ...



n	número de experimentos	algoritmo	promedio de vértices insertados
100	1000	BFS	198
100	1000	BFSDoubleEnded1	102
100	1000	BFSDoubleEnded2	101
250	1000	BFS	499
250	1000	BFSDoubleEnded1	252
250	1000	BFSDoubleEnded2	251
500	1000	BFS	998
500	1000	BFSDoubleEnded1	502
500	1000	BFSDoubleEnded2	501

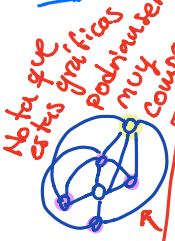
Cuadro 1: Experimentación con árboles binarios



Experimentación con gráficas generadas en Sage

En estos experimentos lo que queremos medir es comparar el tiempo de ejecución entre las estrategias, eso lo hacemos contando el número de elementos que ingresan a la cola en cada algoritmo.

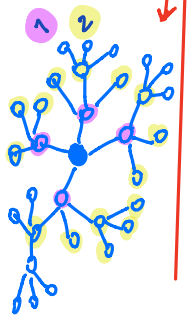
Las dos estrategias se probaron en dos tipos de gráficas. El primer tipo son árboles binarios aleatorios de tamaño $2n + 1$. Esos experimentos consisten en lo siguiente: se escoge un árbol binario, después se escogen los vértices iniciales y finales aleatoriamente. Los resultados de estos experimentos se muestran en la Tabla 1



El segundo tipo de gráficas son las gráficas aleatorias con un grado fijo por vértice, es decir gráficas aleatorias G de tamaño n tal que $\forall v \in V(G) : deg(v) = k$. Los resultados de estos experimentos se muestran en la Tabla 2

Los resultados de los experimentos muestran que sí hay una mejora notable entre usar el algoritmo BFS y el usar alguna de las estrategias del algoritmo BFSDoubleEnded, pero, sorprendentemente, no hay una diferencia notable entre las dos estrategias.

=Muy buen trabajo= ∎



2. En clase vimos una gráfica que requiere $O(2^{(\lfloor \frac{n}{2} \rfloor)})$ iteraciones para que Dijkstra genere el camino más corto entre s y el resto de los vértices. En este problema explorarás las razones detrás de este hecho. Abajo, en la Figura 1 se muestra la familia de gráficas que vimos en clase, el vértice marcado con una flecha roja es s, la gráfica con $n = 5$ se muestra también. Ejecuta Dijkstra en la gráfica con $n = 5$, cuenta el número de rondas que requiere hasta generar los caminos más cortos, intenta encontrar las razones de tal comportamiento. ¿Puede tu razonamiento extenderse para la gráfica de cualquier tamaño?

La ejecución del algoritmo de Dijkstra para $n = 5$ es la de la figura dos:

Ahora para ver las razones por la cual el algoritmo toma tiempo exponencial primero notemos lo siguiente:

1 Si un vértice v se inserta a la cola, entonces sus vecinos de salida también lo hacen después de que insertar a v.

2 En cualquier instante de la ejecución del algoritmo, la distancia de los vértices está ordenada en orden ascendiente de derecha a izquierda, el vértice más a la derecha tiene la menor distancia y el vértice más a la izquierda la mayor.

n	$deg(v)$	número de experimentos	algoritmo	promedio de vértices insertados
100	5	10000	BFS	35.7
100	5	10000	BFSDoubleEnded1	15.6
100	5	10000	BFSDoubleEnded2	19.0
250	5	1000	BFS	91.4
250	5	1000	BFSDoubleEnded1	26.8
250	5	1000	BFSDoubleEnded2	38.6
500	5	1000	BFS	181.9
500	5	1000	BFSDoubleEnded1	75.0
500	5	1000	BFSDoubleEnded2	39.1
100	8	10000	BFS	45.3
100	8	10000	BFSDoubleEnded1	19.7
100	8	10000	BFSDoubleEnded2	17.4
250	8	1000	BFS	116.8
250	8	1000	BFSDoubleEnded1	35.9
250	8	1000	BFSDoubleEnded2	29.1
500	8	1000	BFS	235.4
500	8	1000	BFSDoubleEnded1	52.8
500	8	1000	BFSDoubleEnded2	42.6
100	11	10000	BFS	47.6
100	11	10000	BFSDoubleEnded1	19.6
100	11	10000	BFSDoubleEnded2	17.7
250	11	1000	BFS	245.3
250	11	1000	BFSDoubleEnded1	33.5
250	11	1000	BFSDoubleEnded2	29.7
500	11	1000	BFS	125.4
500	11	1000	BFSDoubleEnded1	47.3
500	11	1000	BFSDoubleEnded2	42.9

Cuadro 2: Experimentación con gráficas aleatorias

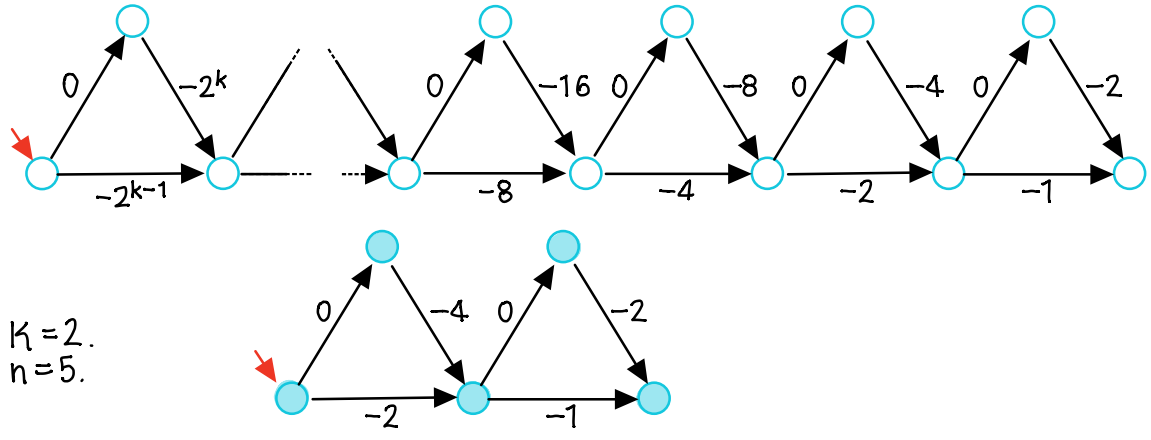


Figura 4: Dijkstra requiere $O(2^{(\lfloor \frac{n}{2} \rfloor)})$ iteraciones.

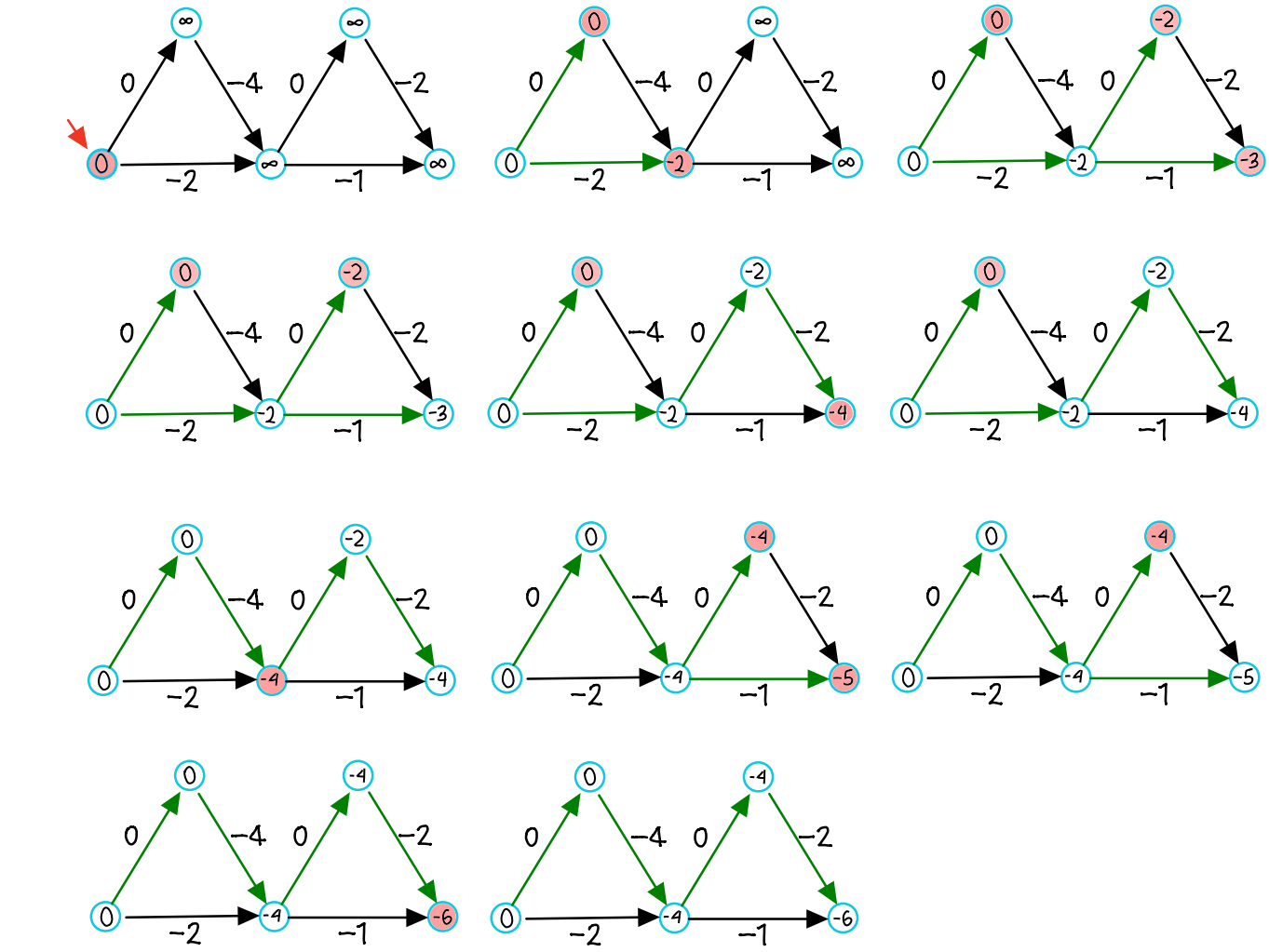


Figura 5: Ejecución del algoritmo de Dijkstra para $n = 5$

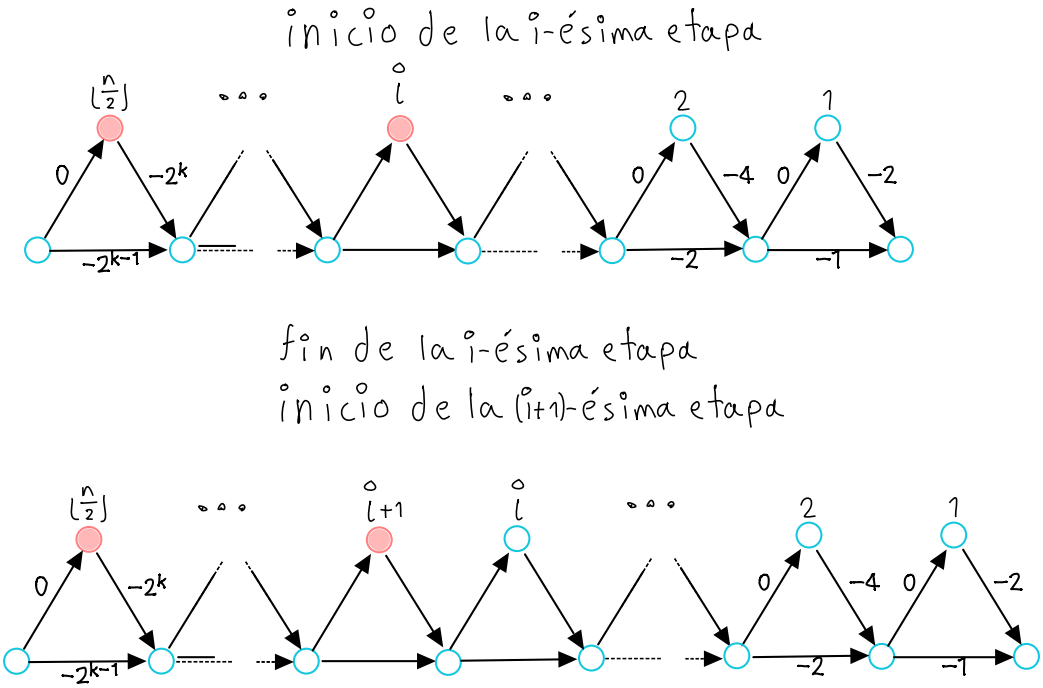
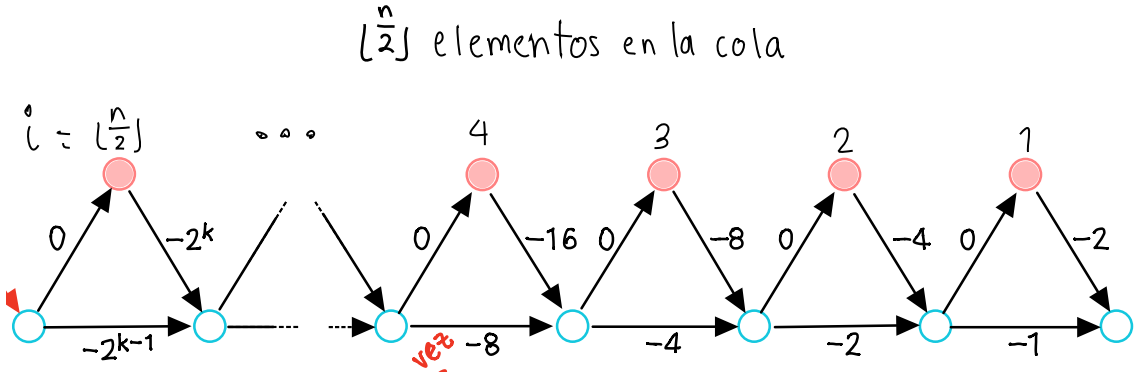


Figura 6: Etapas del algoritmo

Después de $\lceil \frac{n}{2} \rceil$ iteraciones del ciclo while del algoritmo (toma $\lceil \frac{n}{2} \rceil$ iteraciones recorrer todo el camino que forman los vértices de abajo de la gráfica, ya que en esas primeras iteraciones solo vamos removiendo de la cola los vértices que están abajo de la gráfica) , la gráfica sobre la que se está ejecutando el algoritmo se ve de esta manera, solo los vértices que están en la parte de arriba del triangulo están el la cola.



Ahora enumeremos cada vértice que está en la parte de arriba con el índice $\lfloor \frac{n}{2} \rfloor \geq i \geq 1$, como en la imagen de arriba. Llamemos el inicio de una etapa a cuando removemos por primera ves el i-ésimo vértice de arriba, cuando removemos por primera ves el primer i-ésimo vértice de arriba, todos los vértices que están a su derecha en el dibujo ya no estan en la cola, porque todos esos vértices tuvieron que ser agregados a la cola con una distancia menor a ese i-ésimo vértice. Después de $\frac{n}{2}$ etapas termina el algoritmo.

Ahora para el comportamiento exponencial del algoritmo, veamos que pasa en la $(i+1)$-ésima etapa (Figura 6). Sea $f(i)$ el número de iteraciones que toma concluir la i-ésima etapa, en la $(i+1)$-ésima etapa, primero se remueve el $i+1$ vértice de arriba y se inserta su vértice vecino, en la segunda iteración se insertan dos vértices, el i-ésimo vértice de arriba y el vecino de ese i-ésimo vértice.

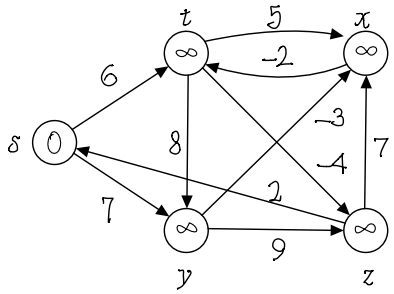
Después el algoritmo remueve el vértice vecino del i-ésimo vértice (el vértice x de la figura 7), eso hará que se tengan que actualizar todos sus vértices hacia la derecha, y eso hará que todos esos vértices se tengan que volver a insertar y sacar de la cola (eso pasa por los puntos (1) y (2)). El tiempo que tarda que tanto x como todos sus vértices hacia la derecha salgan de la cola es $f(i) - 1$ iteraciones, después de esas $f(i) - 1$ iteraciones, solo falta remover de la cola el i-ésimo vértice y dejar fuera de la cola a todos sus vértices que estén a su derecha para concluir la $(i + 1)$-ésima etapa, ese proceso es exactamente el mismo que el de la i-ésima etapa, así remover ese i-ésimo vértice toma tiempo $f(i)$, en total la $(i+1)$-ésima etapa toma tiempo $2f(i) + 1$, el doble de la etapa anterior. Eso explica que el algoritmo tome $O(2^{(\lfloor \frac{n}{2} \rfloor)})$ iteraciones.

■

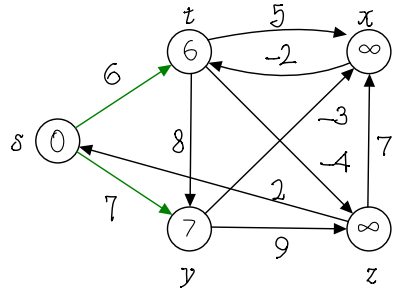
3. Ejecuta el algoritmo de Bellman-Ford en la gráfica de la Figura 8, usando a z como el vértice origen. Ejecuta nuevamente el algoritmo, ahora usando a s como origen pero cambia el peso de la arista (z, x) a 4.

La implementación con programación dinámica del algoritmo de Bellman-Ford por el momento no verifica si hay ciclos negativos. Para verificar si hay un ciclo negativo necesitamos, después de ejecutar BellmanFordDP(s) verificar si podemos generar otro camino más corto de longitud $|V|$, eso lo hacemos iterando nuevamente sobre todos los vértices v y todas las aristas entrantes $u \to v$, y si se cumple que $dist[V - 1, v] > dist[i - 1, u] + w(u \to v)$, entonces podemos formar un camino más corto de s a v con longitud n, eso implica que hay un ciclo negativo. Quedaría el nuevo algoritmo de la siguiente forma

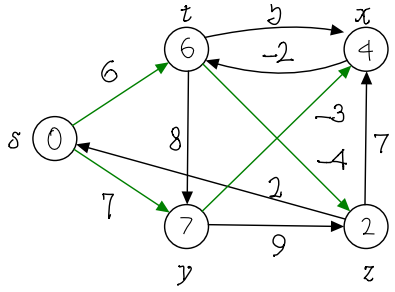
```
BellmanFordDPNegCycles(s):
    líneas del algoritmo BellmardFordDP(s)
    for every vertex v
        for every edge u → v
          if dist[V − 1, v] > dist[V − 1, u] + w(u → v)
            return "Negative Cycle"
```



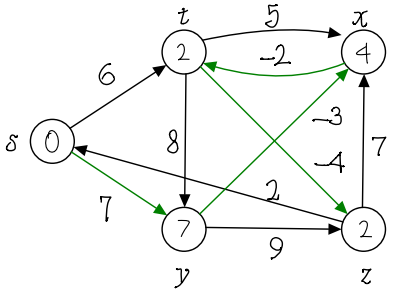
dist[i][v]	s	t	y	x	z
0	0	∞	∞	∞	∞
1					
2					
3					
4					



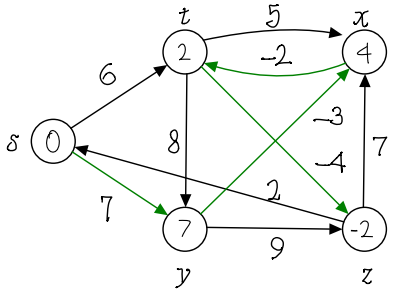
dist[i][v]	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	6	7	∞	∞
2	0				
3	0				
4	0				



dist[i][v]	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	6	7	∞	∞
2	0	6	7	4	2
3	0				
4	0				



dist[i][v]	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	6	7	∞	∞
2	0	6	7	4	2
3	0	2	7	4	2
4	0				



dist[i][v]	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	6	7	∞	∞
2	0	6	7	4	2
3	0	2	7	4	2
4	0	2	7	4	-2

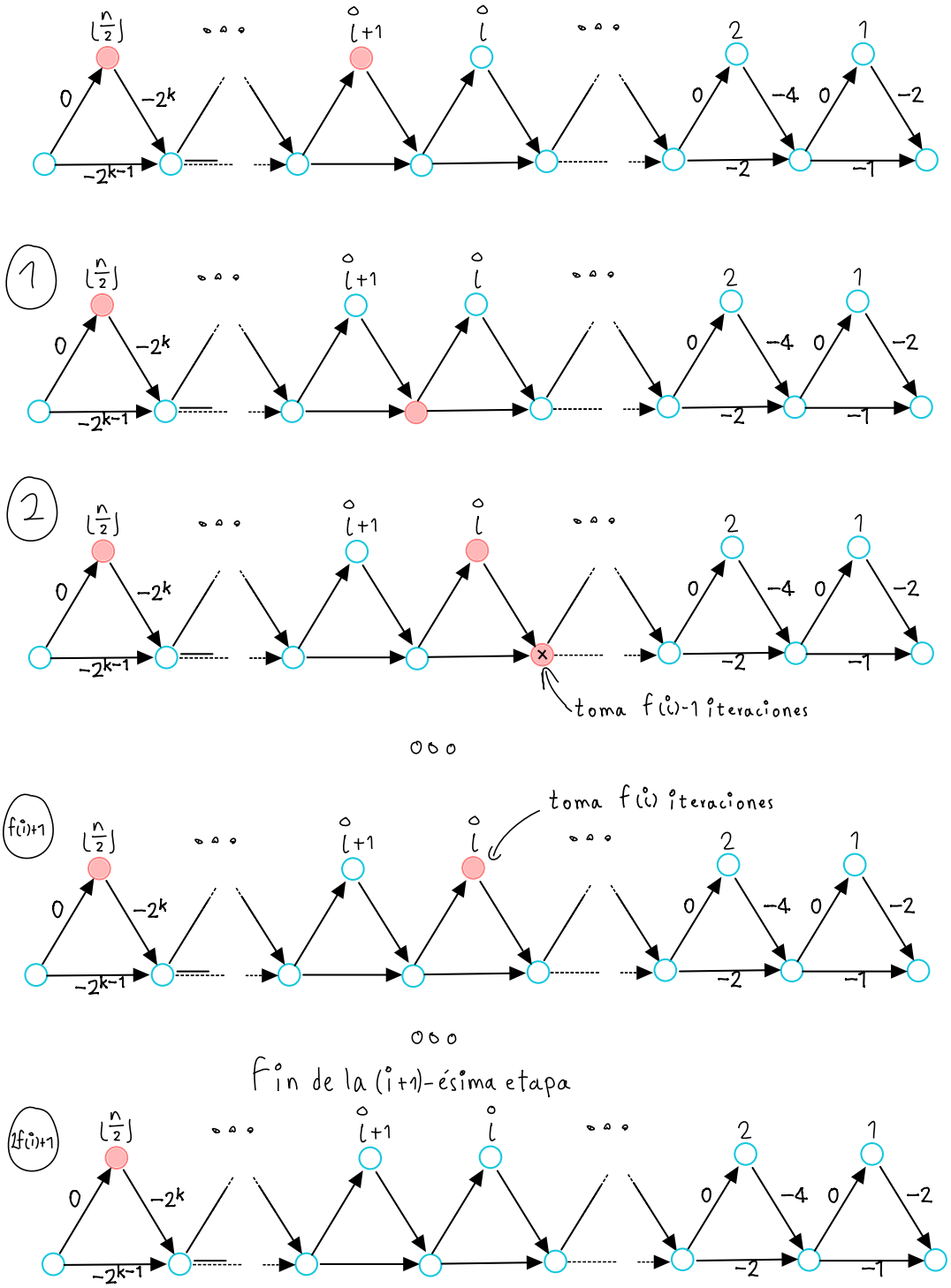
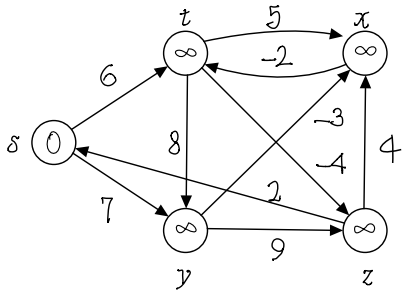


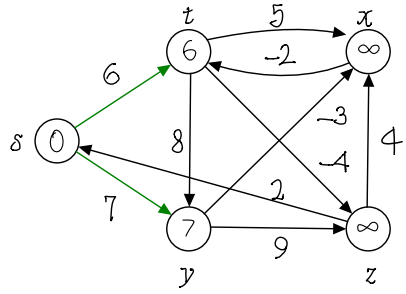
Figura 7: Etapa $(i+1)$-ésima del algoritmo



dist[i][v]	s	t	y	x	z
0	0	∞	∞	∞	∞
1					
2					
3					
4					

v

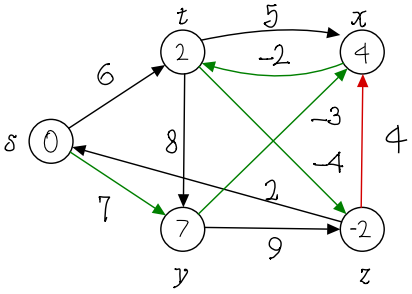
i



dist[i][v]	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	6	7	∞	∞
2	0				
3	0				
4	0				

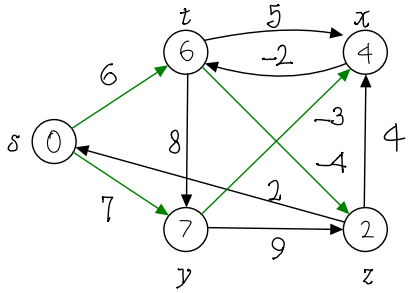
v

i



$dist[4][x] > dist[4][z] + w(z \to x)$

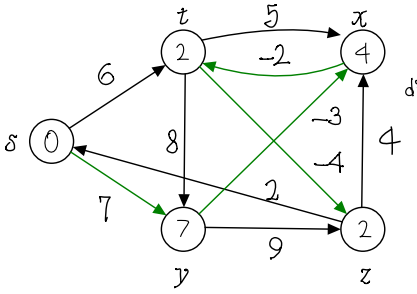
Ciclo negativo



dist[i][v]	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	6	7	∞	∞
2	0	6	7	4	2
3	0				
4	0				

v

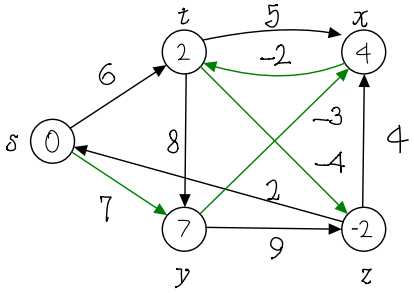
i



dist[i][v]	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	6	7	∞	∞
2	0	6	7	4	2
3	0	2	7	4	2
4	0				

v

i



dist[i][v]	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	6	7	∞	∞
2	0	6	7	4	2
3	0	2	7	4	2
4	0	2	7	4	-2

v

i

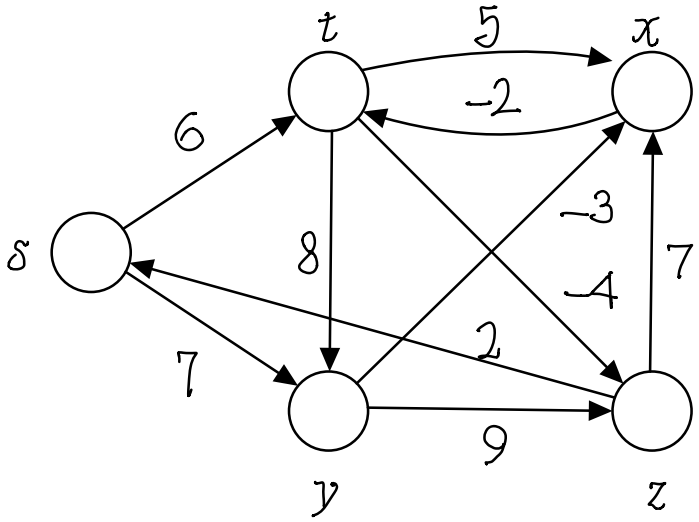


Figura 8: Ejecuta Bellman-Ford.

■