Tarea #2 - Implementación eficiente de Dijkstra

27 de octubre de 2020 - Gonzalo Navarro & Bernardo Subercaseaux

El algoritmo de Dijkstra para caminos más cortos en grafos pesados es uno de los algoritmos más famosos (y prácticos!) de las ciencias de la computación. Su implementación está basada en colas de prioridad, y su eficiencia está por tanto determinada por la eficiencia, en distintas operaciones, de la cola de prioridad subyacente. En esta tarea experimentarán dos formas de implementar una cola de prioridad, y su efecto en el rendimiento del algoritmo de Dijkstra.

Como resultado de la tarea deberán entregar un informe y todo el código utilizado. La corrección estará centrada en el informe. El informe NO debe responder directamente a los problemas planteados en este enunciado en formato *Respuesta 1, Respuesta 2, Respuesta 3, ...*, sino que debe escribirse como un artículo breve, que comienza con un abstract (describe el tema tratado en la tarea y resume los resultados y conclusiones obtenidos) y en cuyo desarrollo se encuentran respondidas todas las preguntas planteadas en el enunciado.

1. Problema a tratar - Caminos más cortos

Sea $G = (V, E)$ un grafo simple y no dirigido, donde cada arista e tiene un peso $w(e) > 0$. Se define el peso de un camino como la suma de los pesos de sus aristas. Naturalmente, un *camino más corto* entre dos nodos u y v es un camino de peso mínimo entre ellos, y su peso se denota $\text{dist}(u, v)$. El problema de *Single Source Shortest Paths* (SSSP) consiste en, dado un nodo inicial s, calcular el peso de los caminos más cortos entre s y todo el resto de los nodos.

Problema:	Single Source Shortest Paths (SSSP)
Input:	Un grafo pesado $G = (V, E)$, un nodo s
Output:	Una lista $d[1..\|V\|]$, donde $d[i] = \text{dist}(s, i)$.

El algoritmo de Dijkstra resuelve precisamente este problema. Antes de describirlo, definiremos una interfaz abstracta para colas de prioridad, la estructura de datos esencial para el algoritmo. Una cola de prioridad Q mantiene una lista de pares elemento-clave (x_i, k_i), donde las claves no son necesariamente únicas pero los elementos sí. La clave asociada a un elemento representa intuitivamente su *prioridad* para ser removido de la cola.[1] Una cola de prioridad debe permitir las siguientes cuatro operaciones:

- **extract-min**(). Esta operación remueve de Q (y retorna) un par (x, k) que minimiza k.
- **insert**(x, k). Esta operación inserta el par (x, k) en Q.
- **empty**(). Esta operación simplemente dice si la cola está vacía, es decir, si $|Q| = 0$.
- **decrease-key**(x, k'). Esta operación cambia la clave del elemento x de k (su clave previa) a k', con $k' \leq k$.

[1] Si bien es natural extraer primero los elementos de máxima prioridad, es común optar por lo contrario a través de una operación `extract-min`. Esto facilita la implementación de Dijkstra, donde la cola de prioridad se usa para obtener los nodos de menor distancia con respecto al nodo inicial.

En esta tarea se compararán dos variantes de colas de prioridad.

- **Montículo binario**: Puede encontrar una descripción (y visualización) de las operaciones en este enlace.
- **Montículo de Fibonacci**: Puede encontrar una descripción (y visualización) de las operaciones en este enlace. Algunas operaciones sobre esta estructura se discutirán en clases, y aparecen en las páginas 62-64 del apunte del curso.

Si bien en los enlaces anteriores aparece código de ejemplo, al igual que puede encontrarse en internet, su implementación debe ser original. Es decir, puede leer tanto como quiera sobre cómo funcionan las diferentes operaciones, pero no debe copiar código ajeno.

1. (0.6 pts.) Compare en una tabla la complejidad de las distintas operaciones en las distintas implementaciones, tanto en términos de peor caso (por operación) como en análisis amortizado. Incluya en el anexo demostraciones (puede guiarse por las del apunte, pero debe escribirlas en sus palabras) para estas complejidades.

2. (1.5 pts.) Programe ambas variantes, con las cuatro operaciones requeridas. Asuma por simplicidad que al construir la cola se pasa un parámetro n que representa el tamaño del universo de elementos. Los elementos son valores en el rango $\{1, \ldots, n\}$. Así, puede mantener en la cola de prioridad un arreglo $P[1..n]$ donde $P[i]$ guarda un (puntero al) nodo que contiene al par (i, k). Esto facilitará la implementación de `decrease-key`, ya que en general cambiar la prioridad de un elemento implica mover su nodo asociado en el montículo, y gracias al arreglo P, puede accederse en tiempo constante al nodo deseado.

Ahora podemos describir el algoritmo de Dijkstra. Este algoritmo tiene varias implementaciones, y algunas no utilizan la operación `decrease-key`, por lo que es fundamental que implemente la versión descrita a continuación, y no una cualquiera de internet.

Algoritmo de Dijkstra (ilustrado en la Figura 1) Dado un grafo $G = (V, E)$ y un nodo inicial s, el algoritmo mantiene una lista $d[1..|V|]$ de distancias, donde se inicializa $d[s] = 0$ y $d[u] = \infty$ para los nodos distintos de s. Se inicializa una cola de prioridad Q vacía. Luego se insertan en Q los pares $(u, d[u])$ por cada nodo u. Es decir, la prioridad de cada nodo corresponde a su distancia (calculada hasta el momento). Se iterará mientras la cola Q no sea vacía. En cada iteración se extrae el nodo de menor prioridad de Q, digamos u, y por cada vecino v de u, se revisa si llegar a v a través de u mejora la distancia a v calculada hasta el momento. Es decir, si $d[v] > d[u] + w(u, v)$, se actualiza $d[v] := d[u] + w(u, v)$, y se utiliza la operación `decrease-key` para disminuir la prioridad de v en Q.

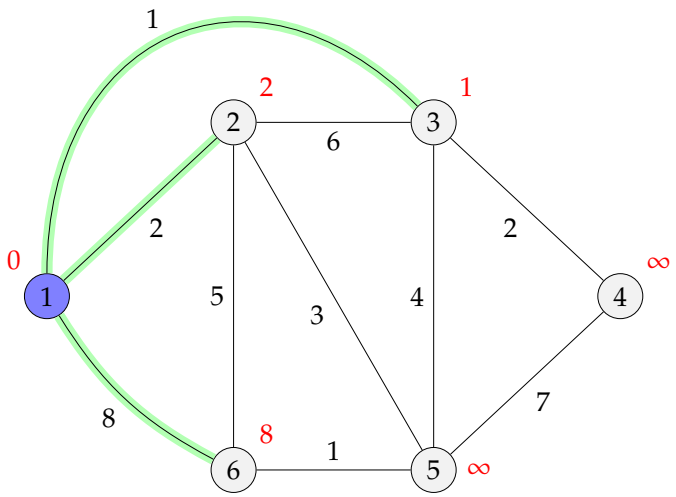

Figura 1: Ejemplo de ejecución del algoritmo de Dijkstra. El nodo inicial s ha sido coloreado en azul. La prioridad de cada nodo, que coincide con la distancia desde s (calculada hasta la iteración actual), aparece en rojo sobre él. Las aristas destacadas en verde han sido exploradas hasta la iteración actual.

3. (0.3 pts.) Escriba el pseudo-código del algoritmo de Dijkstra, utilizando las funciones genéricas de cola de prioridad descritas.

4. (0.5 pts.) Utilizando su análisis de complejidad para las distintas operaciones sobre colas basadas en montículos binarios y de Fibonacci, analice la complejidad total del algoritmo de Dijkstra según el tipo de montículo usado.

5. (0.5 pts.) Programe el algoritmo de Dijkstra utilizando el lenguaje Python (≥ 3.6). Su código debe recibir un argumento por consola indicando si se desea usar una cola de Fibonacci o una cola binaria. Debe recibir un grafo pesado a través de la entrada estándar, luego un nodo inicial s, e imprimir una lista de distancias desde s como resultado.

```
> python dijkstra.py FibonacciHeap
6 10
1 2 2
1 6 8
1 3 1
2 3 6
2 5 3
2 6 5
3 4 2
3 5 4
4 5 7
5 6 1
1
0 2 1 3 5 6 // respuesta
```

El formato del grafo que recibe por entrada estándar se describe a continuación. La primera línea consta de dos enteros: el número de vértices ($|V|$) y el número de aristas($|E|$). A continuación

siguen $|E|$ líneas, cada una de las cuales consiste en tres enteros u, v, w separados por espacios, a interpretarse como que entre los nodos u y v hay una arista de peso w. Los nodos tienen identificadores entre 1 y $|V|$. Finalmente, una última línea contiene el índice del nodo inicial s. Note que el ejemplo de formato corresponde al grafo de ejemplo de la Figura 1.

2. Generación aleatoria de grafos conexos

En esta sección describimos una forma simple para generar grafos aleatorios conexos. Sean $v_1, \ldots, v_n$ los vértices del grafo. Construir las aristas $(v_1, v_2), \ldots, (v_{n-1}, v_n)$ garantiza ya la conexitud. Luego, para cada par de vértices $\{v_i, v_j\}$, se decide independientemente, con una variable aleatoria binaria de peso $0 \leq \rho \leq 1$ (llamada *densidad* del grafo), si se agregará o no una arista entre v_i y v_j. Si se decide agregar la arista, su peso puede ser escogido uniformemente en un intervalo, por ejemplo $[1, 10^9]$.

Recuerde que su informe *debe* describir el proceso de generación de instancias.

6. (0.3 pts.) ¿Cuál es el número esperado de aristas en función de ρ y $|V|$?

3. Hipotésis

7. (0.3 pts.) Describa sus hipótesis sobre el rendimiento de Dijkstra utilizando las distintas colas, en función de la densidad ρ del grafo de entrada. **Explique el razonamiento detrás de sus hipótesis**. No es necesario que las hipótesis sean correctas, solo que sean serias y honestas.

4. Diseño Experimental

Habiendo realizado las hipótesis, el objetivo es diseñar una serie de experimentos que permitan ya sea refutar, verificar o al menos convencerse parcialmente de las hipótesis planteadas.

8. (0.2 pts.) Programe un generador de instancias aleatorias, que se ejecute recibiendo como parámetros un número de nodos $|V|$ y una densidad ρ.

```
> python generator.py  <número de nodos> <densidad>
```

Su programa debe imprimir un grafo conexo, en exactamente el mismo formato descrito en el apartado 5.

4.1. Experimentos

9. (0.3 pts.) Describa valores de $|V|$ y ρ con los que generará experimentos a fin de validar sus hipótesis. Promedie $k = 3$ experimentos para cada conjunto de valores.

5. Resultados

10. (0.7 pts.) Grafique claramente los resultados de sus experimentos. No debería incluir más de 6 gráficos (seleccione los que mejor manifiestan las ideas que quiere comunicar) en el cuerpo de su informe, pero puede incluir gráficos menos importantes en el anexo.

11. (0.3 pts.) Contraste los resultados obtenidos con las hipótesis planteadas.

6. Conclusión

12. (0.4 pts.) Tras contrastar los resultados con las hipótesis planteadas, concluya sobre los resultados obtenidos. Una conclusión no consiste en repetir lo ocurrido, sino en darle sentido y perspectiva. Idealmente la conclusión hace sentido a su lector *después* de haber leído el resto del documento.