J.P. Vidocevich / I. Sánchez
Facultad de Ingeniería y Ciencias Hídricas
Universidad Nacional del Litoral

GTP 2 — Guía de Trabajos Prácticos 2 - Programación 2/3

1. Correspondencias

1. **map2list**. Escribir una función que dado un mapa (diccionario) `M` retorna las listas de claves y valores. Ejemplo: si `M={1->2, 3->5, 8->20}`, entonces debe retornar `Keys = [1,3,8]` y `Vals = [2,5,20]`.

2. **list2map**. Escribir una función que dadas las listas de claves `Keys = [k1,k2,k3, ...]` y valores `Vals, [v1,v2,v3, ...]` retorna el diccionario M con las asignaciones correspondientes `{k1->v1, k2->v2, k3->v3, ...}`. Utilice la signatura `list2map(M, Keys, Vals)`.

   Nota: si hay claves repetidas, sólo debe quedar la asignación correspondiente a la última clave en la lista. Si hay menos valores que claves utilizar cero como valor. Si hay más valores que claves, ignorarlos.

3. **InverseMaps**. Dos correspondencias M_1 y M_2 son inversas una de la otra si tienen el mismo número de asignaciones y para cada par de asignación $x \to y$ en M_1 existe el par $y \to x$ en M_2. Escribir un predicado `areinverse(M_1, M_2)` que determina si las correspondencias M_1 y M_2 son inversas.

4. **mergeMap**. Dadas dos correspondencias A y B, que asocian enteros con listas ordenadas de enteros, escribir una función `mergeMap(A,B,C)` que devuelve en C una correspondencia que asigna al elemento x la fusión ordenada de las dos listas $A[x]$ y $B[x]$. Si x no es clave de A, entonces $C[x]$ debe ser $B[x]$ y viceversa. Sugerencia: utilice la función merge implementada en uno de los ejercicios anteriores.

5. **cutoffMap**. Implemente una función `cutoffMap(M, p, q)` que elimina todas las claves que NO están en el rango $[p, q)$. En las asignaciones que quedan también debe eliminar los elementos de la lista que no están en el rango. Si la lista queda vacía entonces la asignación debe ser eliminada. Por ejemplo: si $M = 1->(2,3,4), 5->(6,7,8), 8->(4,5), 3->(1,3,7)$, entonces `cutoffMap(M,1,6)` debe dejar $M = 1->(2,3,4), 3->(1,3)$. Notar que la clave 5 ha sido eliminada si bien está dentro del rango porque su lista quedaría vacía. Restricciones: el programa no debe usar contenedores auxiliares.

6. **Aplica**. Escribir una función `applyMap(L,M,ML)` que, dada una lista L y una correspondencia M retorna por ML una lista con los resultados de aplicar M a los elementos de L. Si algún elemento de L no está en el dominio de M, entonces el elemento correspondiente de ML no es incluido. Por ejemplo, si $L = (1,2,3,4,5,6,7,1,2,3)$ y $M = (1,2),(2,3),(3,4),(4,5),(7,8)$, entonces, después de hacer `applyMap(L,M,ML)`, debe quedar $ML = (2,3,4,5,8,2,3,4)$.

7. **Camino**. Implemente el predicado `esCamino(G, L)` que recibe una lista L y determina si es o no camino en el grafo G. El grafo se representa como un mapa que relaciona cada vértice (clave) con la lista de sus vértices adyacentes (valor).

8. **Componentes Conexas**. Dado un grafo G encontrar los subconjuntos del mismo D que están desconectados, es decir, los conjuntos de vértices de cada una de las componentes conexas. Por ejemplo, si $G = 1->2, 2->1, 3->4, 4->3$, entonces debe retornar $D = (1,2,3,4)$. La signatura de la función a implementar es `componentesConexas(G, D)`.

9. **Camino Hamiltoniano**. Dado un grafo G y una lista de vértices L implementar `isHamiltonian(G, L)` que determina si L es un camino hamiltoniano en G.

10. **Distancia**. Dado un grafo G y un vértice de partida x se desea determinar la distancia éste al resto de los vértices en G. Se solicita retornar una estructura de capas de vecinos de G alrededor de x definida de la siguiente forma: la capa 0 es x, la capa 1 son los vecinos de x. A partir de allí la capa $n \geq 2$ está formada por los vecinos de los vértices de la capa $n-1$ (es decir la adyacencia de la capa) pero que no están en las capas anteriores (en realidad basta con verificar que no estén en las capas $n-1$ ni $n-2$). Notar que los vértices en la capa n se encuentran a una distancia n del vértices de partida.

11. **Dijkstra**. Programe `Dijkstra( G, u, v, path)` que dado el grafo ponderado G definido como `graphW`, implemente el algoritmo de Dijkstra para retornar el costo del camino más corto entre el vértice de partida u y el vértice de llegada v. Además debe devolver en path uno de los posibles caminos. Si no hay camino posible, retornar un número muy grande (infinito).