Linguaggi di Programmazione Anno Accademico 2023-2024
Progetto febbraio 2024 (E2P)

“Single Source Shortest Paths”

Marco Antoniotti, Fabio Sartori

Consegna:
domenica 25 febbraio 2024, ore 23:59 GMT+1 Time

Introduzione

Calcolare il percorso più breve da un punto a un altro di una mappa[1] è un problema più che noto. Vi sono diversi algoritmi in grado di risolvere questo problema, noto in letteratura come il “*Single Source Shortest Path Problem*” (SSSP Problem, cfr., [CLR+09] capitolo 24).

Lo scopo di questo progetto è l’implementazione dell’algoritmo di Dijkstra (cfr., [CLR+09] 24.3), che risolve il problema SSSP per *grafi diretti* e *connessi* con distanze tra vertici non negative[2].

Per procedere all’implementazione di quest’algoritmo è necessario – e, di fatto, è la parte principale del progetto – produrre un’implementazione di un MINHEAP (o MINPRIORITYQUEUE).

Nel seguito troverete le specifiche dell’API richiesta e dei suggerimenti su come affrontare e risolvere alcuni problemi implementativi che, si pensa, potrebbero presentarsi.

Grafi in Prolog

In “The Art of Prolog” si suggerisce come rappresentare dei semplici grafi diretti in Prolog. L’idea è di inserire direttamente nella base-dati del sistema, fatti del tipo:

```
vertex(v).
vertex(u).
vertex(w).

edge(u, v, 0).
edge(u, w, 10).
edge(v, w, 4).
edge(w, u, 1).
```

Inoltre, possiamo pensare di inserire degli altri dati nella base-dati Prolog; ad esempio, possiamo inserire informazioni riguardanti associate ad ogni vertice mediante dei predicati che rappresentano queste associazioni. Ad esempio, potremmo supporre che ad ogni vertice sia associata una posizione su una mappa.

```
pos(v, 10, 22).
pos(u, 234, 11).
pos(w, 1, 34).
```
...

[1] Ad esempio, calcolare la distanza tra Porta Ludovica e Piazza Napoli a Milano (a meno di trovarsi in un racconto di Umberto Eco).
[2] In realtà, il vero vincolo è che non esistano cicli nel grafo dove la somma dei pesi degli archi sia negativa; per il progetto assumiamo che tutti i pesi siano maggiori o uguali a 0.

Una volta scelta una rappresentazione in memoria di un grafo (diretto) è semplice manipolare i grafi in Prolog e costruire delle API che ci permettono di costruire algoritmi più complessi, quali l'algoritmo di Dijkstra per la soluzione del problema SSSP.

Interfaccia Prolog per la manipolazione di grafi

L'interfaccia richiesta è descritta nel seguito. Si noti come ogni predicato mette in relazione un identificatore speciale che denota un particolare grafo. Si noti anche che tutti i predicati richiesti presumono l'utilizzo delle primitive di manipolazione della base di dati Prolog: `assert`, `retractall` e `retract` in particolare[3].

`new_graph(G).`
Questo predicato inserisce un nuovo grafo nella base-dati Prolog. Una sua semplice implementazione potrebbe essere

```
new_graph(G) :- graph(G), !.
new_graph(G) :- assert(graph(G)), !.
```

Esempio

```
?- new_graph(il_mio_grafettino).
true

?- graph(G).
G = il_mio_grafettino
…
```

`delete_graph(G).`
Rimuove tutto il grafo (vertici e archi inclusi) dalla base-dati Prolog.

`new_vertex(G, V).`
Aggiunge il vertice V nella base-dati Prolog. N.B. si richiede che il predicato che rappresenta i vertici, da aggiungere alla base-dati Prolog, sia `vertex(G, V)`. Anche in questo caso dovrete usare i predicati di manipolazione della base-dati Prolog. Notate che è, in generale, permesso che uno stesso vertice possa appartenere a più grafi distinti.

`vertices(G, Vs).`
Questo predicato è vero quanto `Vs` è una lista contenente tutti i vertici di `G`[4].

`list_vertices(G).`
Questo predicato stampa alla console dell'interprete Prolog una lista dei vertici del grafo `G` (usate `listing/1`).

`new_edge(G, U, V, Weight).`
Aggiunge un arco del grafo G alla base dati Prolog. N.B. è richiesto che il predicato che rappresenta gli archi, da aggiungere alla base-dati Prolog, sia `edge(G, U, V, Weight)`. Per comodità potete anche costruire una versione `new_edge/3` così definita:

```
new_edge(G, U, V) :- new_edge(G, U, V, 1).
```

`edges(G, Es).`
Questo predicato è vero quando `Es` è una lista di tutti gli archi presenti in `G`.

`neighbors(G, V, Ns).`
Questo predicato è vero quando `V` è un vertice di `G` e `Ns` è una lista contenente gli archi, `edge(G, V, N, W)`, che portano ai vertici `N` immediatamente raggiungibili da `V`.

[3] Attenzione: i predicati di manipolazione della base-dati Prolog possono lasciare delle alternative negli stack di esecuzione; in questo caso potrebbe essere che il sistema Prolog possa generare delle soluzioni extra. Cercate di evitarle.
[4] Il predicato standard `findall/3` vi sarà utile in questo e altri casi.

`list_edges(G).`
Questo predicato stampa alla console dell'interprete Prolog una lista degli archi del grafo `G` (è il simmetrico di `list_vertices/1`).

`list_graph(G).`
Questo predicato stampa alla console dell'interprete Prolog una lista dei vertici e degli archi del grafo `G`.

SSSP in Prolog

La soluzione del problema SSSP con l'algoritmo di Dijkstra dovrà essere implementata mediante i predicati seguenti. I predicati più delicati da implementare sono quelli che modificano lo stato della base-dati del Prolog; di fatto, si tratta di implementare una modifica dello "stato" della memoria del sistema. L'API per la soluzione del problema SSSP è la seguente.

`distance(G, V, D).`
Questo predicato è vero quando `V` è un vertice di `G` e, durante e dopo l'esecuzione dell'algoritmo di Dijkstra, la distanza minima del vertice `V` dalla "sorgente" è `D` (cfr., [CLR+09] 24.3).

`visited(G, V).`
Questo predicato è vero quando `V` è un vertice di `G` e, durante e dopo l'esecuzione dell'algoritmo di Dijkstra, `V` risulta "visitato" (cfr., [CLR+09] 24.3).

`previous(G, V, U).`
Questo predicato è vero quando `V` ed `U` sono vertici di `G` e, durante e dopo l'esecuzione dell'algoritmo di Dijkstra, il vertice `U` è il vertice "precedente" a `V` nel cammino minimo dalla "sorgente" a `V` (cfr., [CLR+09] 24.3).

NB i predicati `distance/3`, `visited/2` e `previous/3` sono "dinamici" e sono asseriti (o ritrattati) durante l'esecuzione dell'algoritmo. I prossimi predicati servono per modificare queste "memorizzazioni".

`change_distance(G, V, NewDist).`
Questo predicato ha sempre successo con due effetti collaterali: prima tutte le istanze di `distance(G, V, _)` sono ritirate dalla base-dati Prolog, e quindi `distance(G, V, NewDist)` è asserita.

`change_previous(G, V, U).`
Questo predicato ha successo con due effetti collaterali: prima tutte le istanze di `previous(G, V, _)` sono ritirate dalla base-dati Prolog, e quindi `previous(G, V, U)` è asserita.

I predicati che servono per risolvere il problema SSSP sono `dijkstra_sssp/1` e `shortest_path/3`.

`dijkstra_sssp(G, Source).`
Questo predicato ha successo con un effetto collaterale. Dopo la sua prova, la base-dati Prolog ha al suo interno i predicati `distance(G, V, D)` per ogni `V` appartenente a `G`; la base-dati Prolog contiene anche i predicati `previous(G, V, U)` e `visited(V)` per ogni `V`, ottenuti durante le iterazioni dell'algoritmo di Dijkstra. Naturalmente i predicati `distance(G, V, D)` e `previous(G, V, U)` devono essere corretti rispetto alla soluzione del problema SSSP.

`shortest_path(G, Source, V, Path).`
Questo predicato è vero quando Path è una lista di archi

```
[edge(G, Source, N1, W1),
 edge(G, N1, N2, W2),
 …,
 edge(G, NK, V, Wk)]
```

che rappresenta il "cammino minimo" da `Source` a `V`.

Una tipica interrogazione del sistema potrebbe essere la seguente:

```
?- dijkstra_sssp(my_graph, s),
   shortest_path(my_graph, s, v42, ShortestPath).
```

Come anticipato, l'implementazione dell'algoritmo di Dijkstra ha bisogno di un'implementazione funzionante di una coda a priorità (*priority queue*), in altre parole di un MINHEAP. Nel seguito si descriverà l'API di una libreria Prolog che implementa un MINHEAP.

MINHEAP in Prolog

Un MINHEAP è una struttura dati che prevede le sequenti operazioni: NEWHEAP, INSERT, HEAD, EXTRACT, MODIFYKEY. Si rimanda a [CLR+09] Capitolo 6 e Sedgewick e Wayne *Algorithms* [SW11] Capitolo 2.4 per la spiegazione del funzionamento di queste operazioni.

La libreria Prolog che implementa il MINHEAP avrà l'API seguente.

`new_heap(H).`
Questo predicato inserisce un nuovo heap nella base-dati Prolog. Una sua semplice implementazione potrebbe essere

```
new_heap(H) :- heap(H, _S), !.
new_heap(H) :- assert(heap(H, 0)), !.
```

Notate che il predicato `heap(H, S)` mantiene la dimensione corrente dello heap nel secondo argomento.

`delete_heap(H).`
Rimuove tutto lo heap (incluse tutte le "entries") dalla base-dati Prolog.

`heap_size(H, S).`
Questo predicato è vero quanto `S` è la dimensione corrente dello heap. Una semplice implementazione potrebbe essere:

```
heap_size(H, S) :- heap(H, S).
```

`empty(H).`
Questo predicato è vero quando lo heap `H` non contiene elementi.

`not_empty(H).`
Questo predicato è vero quando lo heap `H` contiene almeno un elemento.

Un MINHEAP mantiene delle associazioni tra chiavi `K` e valori `V`. Si suggerisce di mantenere queste associazioni nella base-dati Prolog mediate predicati `heap_entry(H, P, K, V)`, dove `P` è la "posizione" nello heap `H`. I fatti `heap_entry/4` dovranno essere asseriti e ritrattati a seconda della bisogna durante le operazioni di ristrutturazione del MINHEAP.

`head(H, K, V).`
Il predicato `head/3` è vero quando l'elemento dello heap `H` con chiave minima `K` è `V`.

`insert(H, K, V).`
Il predicato `insert/3` è vero quando l'elemento `V` è inserito nello heap `H` con chiave `K`. Naturalmente, lo heap `H` dovrà essere ristrutturato in modo da mantenere la proprietà che `head(H, HK, HV)` sia vero per `HK` minimo e che la "heap property" sia mantenuta ad ogni nodo dello heap.

`extract(H, K, V).`
Il predicato `extract/3` è vero quando la coppia `K`, `V` con `K` minima, è rimossa dallo heap `H`. Naturalmente, lo heap `H` dovrà essere ristrutturato in modo da mantenere la proprietà che `head(H, HK, HV)` sia vero per `HK` minimo e che la "heap property" sia mantenuta ad ogni nodo dello heap.

`modify_key(H, NewKey, OldKey, V).`
Il predicato `modify_key/4` è vero quando la chiave `OldKey` (associata al valore `V`) è sostituita da `NewKey`. Naturalmente, lo heap `H` dovrà essere ristrutturato in modo da mantenere la proprietà

che `head(H, HK, HV)` sia vero per `HK` minimo e che la "heap property" sia mantenuta ad ogni nodo dello heap.

`list_heap(H).`
Il predicato richiama `listing/1` per stampare sulla console Prolog lo stato interno dello heap.

Il consiglio è di implementare la libreria MINHEAP, e di assicurarsi che funzioni, prima di passare ad implementare l'algoritmo di Dijkstra.

Common Lisp

L'implementazione in Common Lisp richiede, come immaginabile, l'utilizzo di assegnamenti per modificare lo stato del sistema[5].

Grafi in Common Lisp

Vi sono diversi modi di rappresentare i grafi in Common Lisp. Naturalmente è possibile adottare le rappresentazioni standard a *lista di adiacenza* (*adjacency list*) o a *matrice di adiacenza* (*adjacency matrix*), ma si cercherà di adottare una rappresentazione ibrida più simile alla rappresentazione Prolog per, si spera, semplificare il lavoro.

L'idea principale è di avere dei vertici rappresentati da *atomi* (*simboli* e *numeri interi*) e di definire delle *hash-tables* che useranno questi atomi come chiavi. De facto, queste hash-tables si comporteranno come la knowledge base di Prolog[6].

Di conseguenza assumiamo di avere le seguenti hash-tables (si vedrà dopo come crearle e manipolarle).

```
*vertices*
*edges*
*graphs*
*visited*
*distances*
*previous*
```

La convenzione di usare asterischi attorno ai nomi è una pura convenzione dei programmatori Common Lisp.

Hash Tables in Common Lisp

Le hash-tables sono strutture dati primitive in Common Lisp. Le funzioni che le manipolano sono essenzialmente le seguenti.

`make-hash-table`: crea una hash-table.

`clrhash`: rimuove ogni coppia chiave/valore dalla hash-table.

`gethash`: ritorna il valore associato ad una chiave, o NIL.

`remhash:` rimuove il valore associato ad una chiave.

`maphash`: come `mapcar` ma prende una funzione di due argomenti (uno per la chiave ed uno per il valore) ed una hash-table ed applica la funzione ad ognuna delle coppie.

Per modificare il contenuto di una hash-table, oltre a `clrhash`, si usa l'operatore di assegnamento, **`setf`**, in congiunzione con `gethash`.

La seconda parte del capitolo "*Collections*" di "*Practical Common Lisp*" di Seibel contiene un'altra introduzione alle hash-tables (http://www.gigamonkeys.com/book/collections.html).

[5] É possibile costruire queste strutture dati in modo funzionale e senza effetti collaterali, ma risulta tanto complicato quanto elegante.
[6] Non dovrebbe sfuggire l'analogia con le "tabelle" di una base di dati relazionale.

Le Hash Tables dei grafi in Common Lisp per il progetto

Tornando alle hash-tables per il progetto, esse vanno definite nel seguente modo[7].

```
(defparameter *vertices* (make-hash-table :test #'equal))
…
(defparameter *graphs* (make-hash-table :test #'equal))
… ; Etc. Etc. Etc.
```

Il parametro passato per keyword, `:test`, è la funzione che viene usata per testare se un certo elemento è una chiave nella hash-table; nel caso in questione si usa la funzione **`equal`**.

Se si vuole aggiungere un grafo alla hash-table *graphs* si usa il codice qui sotto

```
(setf (gethash 'il-mio-grafettino *graphs*) …)
```

Si pazienti ancora un attimo riguardo il valore che si inserirà nella hash-table.

Interfaccia Common Lisp per la manipolazione di grafi

Seguendo la falsariga dell'interfaccia Prolog, per il Common Lisp si dovrà predisporre l'interfaccia descritta qui sotto.

`is-graph` *`graph-id → graph-id or NIL`*
Questa funzione ritorna il graph-id stesso se questo grafo è già stato creato, oppure NIL se no. Una sua implementazione è semplicmente

```
(defun is-graph (graph-id)
     ;; graph-id è un atomo: un simbolo (non NIL) o un intero.
    (gethash graph-id *graphs*))
```

`new-graph` *`graph-id → graph-id`*
Questa funzione genera un nuovo grafo e lo inserisce nel data base (ovvero nella hash-table) dei grafi. Una sua implementazione potrebbe essere la seguente:

```
(defun new-graph (graph-id)
    ;; graph-id è un atomo: un simbolo (non NIL) o un intero.
    (or (gethash graph-id *graphs*)
        (setf (gethash graph-id *graphs*) graph-id)))
```

Esempio

```
cl-prompt> (new-graph 'il-mio-grafettino)
IL-MIO-GRAFETTINO

cl-prompt> (is-graph 'il-mio-grafettino)
IL-MIO-GRAFETTINO

cl-prompt> (is-graph 'G2)
NIL
```

`delete-graph` *`graph-id`* → NIL
Rimuove l'intero grafo dal sistema (vertici archi etc); ovvero rimuove tutte le istanze presenti nei data base (ovvero nelle hash-tables) del sistema.

`new-vertex` *`graph-id vertex-id → vertex-rep`*
Aggiunge un nuovo vertice *`vertex-id`* al grafo *`graph-id`*. Notate come la rappresentazione di un vertice associ un vertice ad un grafo (o più). Una possibile implementazione di new-vertex potrebbe essere la seguente:

```
(defun new-vertex (graph-id vertex-id)
     ;; graph-id è un atomo: un simbolo (non NIL) o un intero.
```

[7] NB. Ogni volta che si ricarica il codice, le suddette hash-tables sono re-instaziate e tutto quello che contenevano prima, garbage-collected. È un effetto collaterale di `defparameter`.

```
;; vertex-id pure.
(setf (gethash (list 'vertex graph-id vertex-id)
               *vertices*))
      (list 'vertex graph-id vertex-id)))
```

graph-vertices *graph-id → vertex-rep-list*
Questa funzione torna una lista di vertici del grafo.

new-edge *graph-id vertex-id vertex-id* &optional *weight → edge-rep*
Questa funzione aggiunge un arco del grafo graph-id nella hash-table `*edges*`. La rappresentazione di un arco è

```
(edge graph-id u v weight)
```

graph-edges *graph-id → edge-rep-list*
Questo funzione ritorna una lista una lista di tutti gli archi presenti in `graph-id`.

graph-vertex-neighbors *graph-id vertex-id → vertex-rep-list*
Questa funzione ritorna una lista *vertex-rep-list* contenente gli archi,

```
(edge graph-id vertex-id N W),
```

che portano ai vertici `N` immediatamente raggiungibili da `vertex-id`.

graph-print *graph-id*
Questa funzione stampa alla console dell'interprete Common Lisp una lista dei vertici e degli archi del grafo `graph-id`.

SSSP in Common Lisp

Anche in Common Lisp dovrete implementare un'interfaccia standardizzata, sempre sulla falsariga di quella descritta per il sistema Prolog.

L'API per la soluzione del problema SSSP è la seguente.

sssp-dist *graph-id vertex-id → d*
Questa funzione, dato un *vertex-id* di un grafo *graph-id* ritorna, durante e dopo l'esecuzione dell'algoritmo di Dijkstra, la distanza minima *d* del vertice *vertex-id* dalla "sorgente" (cfr., [CLR+09] 24.3).

sssp-visited *graph-id vertex-id → boolean*
Questo predicato è vero quando *vertex-id* è un vertice di *graph-id* e, durante e dopo l'esecuzione dell'algoritmo di Dijkstra, *vertex-id* risulta "visitato" (cfr., [CLR+09] 24.3).

sssp-previous *graph-id V → U*
Questa funzione, durante e dopo l'esecuzione dell'algoritmo di Dijkstra, ritorna il vertice *U* che è il vertice "precedente" a *V* nel cammino minimo dalla "sorgente" a *V* (cfr., [CLR+09] 24.3).

NB le funzioni `dist`, `visited` e `previous` operano sulle hash-tables descritte precedentemente e dal nome eponimo.

sssp-change-dist *graph-id V new-dist* → NIL
Questa funzione ha solo un effetto collaterale: alla chiave

```
(graph-id V)
```

nella hash-table `*distances*` viene associato il valore *new-dist*.

sssp-change-previous *graph-id V U* → NIL
Questa funzione ha solo un effetto collaterale: alla chiave

```
(graph-id V)
```

nella hash-table `*previous*` viene associato il valore *U*.

Le funzioni che servono per risolvere il problema SSSP sono `sssp` e `shortest-path`.

`sssp-dijkstra` *`graph-id source`* → `NIL`
Questa funzione termina con un effetto collaterale. Dopo la sua esecuzione, la hash-table `*distances*` contiene al suo interno le associazioni (*`graph-id V`*) ⇒ *`d`* per ogni `V` appartenente a
`graph-id`; la hash-table `*previous*` contiene le associazioni (*`graph-id V`*) ⇒ *`U`*; infine la hash-table `*visited*` contiene the associazioni *`graph-id V`*) ⇒ `{T, NIL}`. Naturalmente il contenuto delle varie hash-tables predicati deve essere corretto rispetto alla soluzione del problema SSSP.

`sssp-shortest-path` *`G Source V`* → *`Path`*
Questa funzione ritorna una lista di archi

```
((edge G Source N1 W1)
 (edge G N1 N2 W2)
 …
 (edge G NK V Wk))
```

che rappresenta il "cammino minimo" da `Source` a `V`.

Una tipica interazione con il sistema potrebbe essere la seguente:

```
cl-prompt> (sssp-dijkstra 'my-graph 's)
NIL

cl-prompt> (sssp-shortest-path 'my-graph 's 'v42)
((edge MY-GRAPH S N1 3)
 (edge MY-GRAPH N1 N4 12)
 (edge MY-GRAPH N4 V12 1)
 (edge MY-GRAPH V12 V42 7))
```

Come anticipato, l'implementazione dell'algoritmo di Dijkstra ha bisogno di un'implementazione funzionante di una coda a priorità (*priority queue*), in altre parole di un MINHEAP. Nel seguito si descriverà l'API di una libreria Common Lisp che implementa un MINHEAP.

MINHEAP in Common Lisp

Un MINHEAP è una struttura dati che prevede le sequenti operazioni: NEWHEAP, INSERT, HEAD, EXTRACT, MODIFYKEY. Si rimanda a [CLR+09] Capitolo 6 e Sedgewick e Wayne *Algorithms* [SW11] Capitolo 2.4 per la spiegazione del funzionamento di queste operazioni.

La libreria Common Lisp che implementa il MINHEAP avrà l'API descritta nel seguito. Seguendo l'esempio descritto precedentemente per la gestione dei grafi, si assume la presenza di una hash-table chiamata

`*heaps*`

che è costruita nel solito modo.

Per implementare gli heaps in Common Lisp nella maniera più tradizionale è necessario introdurre una seconda struttura dati Common Lisp: gli arrays (ed i vettori).

Per creare un array in Common Lisp si usa la funzione **`make-array`**; dato che useremo solo array monodimensionali la chiamata che serve è semplicemente la seguente:

```
cl-prompt> (make-array 3)
#(NIL NIL NIL) ; In Lispworks.
```

che costruisce un array di `N` elementi (in questo caso 3). Per recuperare un elemento nella posizione i-esima si usa la funzione **`AREF`**; per inserirne uno nella posizione i-esima si usa **`SETF`** in combinazione con `AREF`.

```
cl-prompt> (defparameter a (make-array 3))
A

cl-prompt> a
#(NIL NIL NIL) ; In Lispworks.

cl-prompt> (aref a 1) ; Gli arrays sono indicizzati a 0.
NIL

cl-prompt> (setf (aref a 1) 42)
42

cl-prompt> a
#(NIL 42 NIL)

cl-prompt> (aref a 1)
42
```

Le funzioni da implementare sono le seguenti.

new-heap *H &optional (capacity 42) → heap-rep*
Questa funzione inserisce un nuovo heap nella hash-table `*heaps*`. Una sua semplice implementazione potrebbe essere

```
(defun new-heap (heap-id &optional (capacity 42))
   (or (gethash heap-id *heaps*)
       (setf (gethash heap-id *heaps*)
             (list 'heap heap-id 0 (make-array capacity)))))
```

Quindi una heap-rep è una lista (potete anche usare altri oggetti Common Lisp) siffatta:

`(HEAP` *heap-id heap-size actual-heap*`)`

Ne consegue che anche le funzioni di "accesso" ad uno `heap-rep` sono le ovvie: **heap-id**, **heap-size** e **heap-actual-heap**.

Notate che si usa il "nome" dello heap per recuperarlo[8] Notate che nella hash table ***heaps*** si mantengono le "heap-reps" indicizzate con il nome dello heap.

Nota: potete usare **defstruct** al posto di rappresentare uno heap come una lista `(heap id N <array>)`; l'importante è che l'interfaccia sia rispettata.

Le altre funzioni necessarie sono

heap-delete *heap-id → T*
Rimuove tutto lo heap indicizzato da *heap-id*. Potete usare la funzione **remhash** per questo scopo.

heap-empty *heap-id → boolean*
Questo predicato è vero quando lo heap *heap-id* non contiene elementi.

heap-not-empty *heap-id → boolean*
Questo predicato è vero quando lo heap *heap-id* contiene almeno un elemento.

Un MINHEAP mantiene delle associazioni tra chiavi `K` e valori `V`. L'implementazione degli heap in Common Lisp è la solita basata su un array monodimensionale che si può trovare, ad esempio, in [CLR+09] e [SW11].

heap-head *heap-id → (K V)*
La funzione `heap-head` ritorna una lista di due elementi dove `K` è la chiave minima e `V` il valore associato.

[8] Ciò non è strettamente necessario, ma rende, come si è detto, il codice più simile a quello Prolog.

heap-insert *heap-id K V → boolean*
La funzione `heap-insert` inserisce l'elemento `V` nello heap `heap-id` con chiave `K`. Naturalmente, lo heap `heap-id` dovrà essere ristrutturato in modo da mantenere la "heap property" ad ogni nodo dello heap.

heap-extract *heap-id → (K V)*
La funzione `heap-extract` ritorna la lista con `K`, `V` e con `K` minima; la coppia è rimossa dallo heap `heap-id`. Naturalmente, lo heap `heap-id` dovrà essere ristrutturato in modo da mantenere la "heap property" ad ogni nodo dello heap.

heap-modify-key *heap-id new-key old-key V → boolean*
La funzone `heap-modify-key` sostituisce la chiave `OldKey` (associata al valore `V`) con `NewKey`. Naturalmente, lo heap `heap-id` dovrà essere ristrutturato in modo da mantenere la "heap property" ad ogni nodo dello heap. Attenzione che al contrario dell'implementazione Prolog, questa operazione può risultare molto costosa (lineare nella dimensione dello heap). Perché? Come potreste pensare di aumentare la "heap-rep" per rendere questa operazione più efficiente?

heap-print *heap-id → boolean*
Questa funzione stampa sulla console lo stato interno dello heap `heap-id`. Questa funzione vi serve soprattutto per debugging; il formato di questa stampa è libero.

Anche per il Common Lisp, il consiglio è di implementare la libreria MINHEAP, e di assicurarsi che funzioni, prima di passare ad implementare l'algoritmo di Dijkstra.

Tests

Per essere sicuri di avere degli algoritmi funzionanti, è bene generare una serie di grafi a caso (inclusi quelli presentati in [CLR+09] e [SW11]) e testare che il proprio algoritmo si comporti come ci si aspetta. Inoltre, si consiglia di assicurarsi che la coda con priorità funzioni, *prima* di procedere alla stesura dell'algoritmo SSSP.

Da consegnare

Dovrete consegnare un file `.zip` (i files `.tar` o `.tar.gz` o `.rar` **non sono accettabili!!!**) dal nome

*Cognomi_Nomi_Matricola*_SSSP_LP_202402.zip

Nomi e *Cognomi* devono avere solo la prima lettera maiuscola, *Matricola* deve avere lo zero iniziale se presente.

Questo file *deve contenere una sola directory* (folder, cartella) con lo stesso nome del file (meno il suffisso `.zip`). Al suo interno ci deve essere *una sola* sottodirectory chiamata 'Prolog'. Al suo interno questa directory deve contenere il file Prolog caricabile e interpretabile, più tutte le istruzioni e i commenti che riterrete necessari includere. Il file Prolog si deve chiamare `sssp.pl`. Istruzioni e commenti devono trovarsi in un file chiamato `README.txt`.

Consideriamo, ad esempio, un ipotetico studente Mario Epaminonda Bianchi Rossi, con matricola 012345. Questo studente dovrà consegnare un file di nome `Bianchi_Rossi_Mario_Epaminonda_012345_SSSP_LP_202402.zip` che, una volta spacchettato, dovrà produrre questa struttura di directory e files:

```
Bianchi_Rossi_Mario_Epaminonda_012345_SSSP_LP_202402/
      Lisp/
            sssp.lisp
            README.txt
      Prolog/
            sssp.pl
            README.txt
```

Potete aggiungere altri files, ma il loro caricamento dovrà essere effettuato automaticamente al momento del caricamento ("loading") del file `sssp.pl`.

Come sempre, valgono le direttive standard (reperibili sulla piattaforma Moodle) circa la formazione dei gruppi.

Ogni file deve contenere all'inizio un commento con il nome e matricola di ogni componente del gruppo. Ogni persona deve consegnare un elaborato, anche quando ha lavorato in gruppo.

Il termine ultimo della consegna sulla piattaforma Moodle è domenica 25 febbraio, ore 23:55 GMT+1 Time.

ATTENZIONE!

Non fate copia-incolla di codice da questo documento, o da altre fonti. Spesso vengono inseriti dei caratteri UNICODE nel file di testo che creano dei problemi agli scripts di valutazione.

Valutazione

In aggiunta a quanto detto nella sezione "Indicazioni e requisiti" seguono ulteriori informazioni sulla procedura di valutazione.

Abbiamo a disposizione una serie di esempi e test standard *che verranno eseguiti in maniera completamente automatica* e saranno usati per la valutazione programmi, così da garantire l'oggettività della valutazione. Se i files sorgenti non potranno essere letti/caricati nell'ambiente SWI-Prolog e/o nell'ambiente Lispworks Common Lisp, il progetto non sarà ritenuto sufficiente. (N.B.: il programma *deve necessariamente funzionare in SWI-Prolog e in Lispworks*, ma non necessariamente in ambiente Windows; usate solo primitive presenti nell'ambiente SWI-Prolog e Lispworks).

Il mancato rispetto dei nomi indicati per funzioni e predicati, o anche delle strutture proposte e della semantica esemplificata nel testo del progetto, oltre a comportare ritardi e possibili fraintendimenti nella correzione, possono comportare una diminuzione nel voto ottenuto.

Riferimenti

[CLR+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, Third Edition, The MIT Press, 2009 (Capitoli 6 e 24.3)

[SW11] Robert Sedgewick, Kevin Wayne, *Algorithms*, Fourth Edition, Addison Wesley Professional, 2011 (Capitoli 2.4 `http://algs4.cs.princeton.edu/24pq/` e 4.4 `http://algs4.cs.princeton.edu/44sp/`)