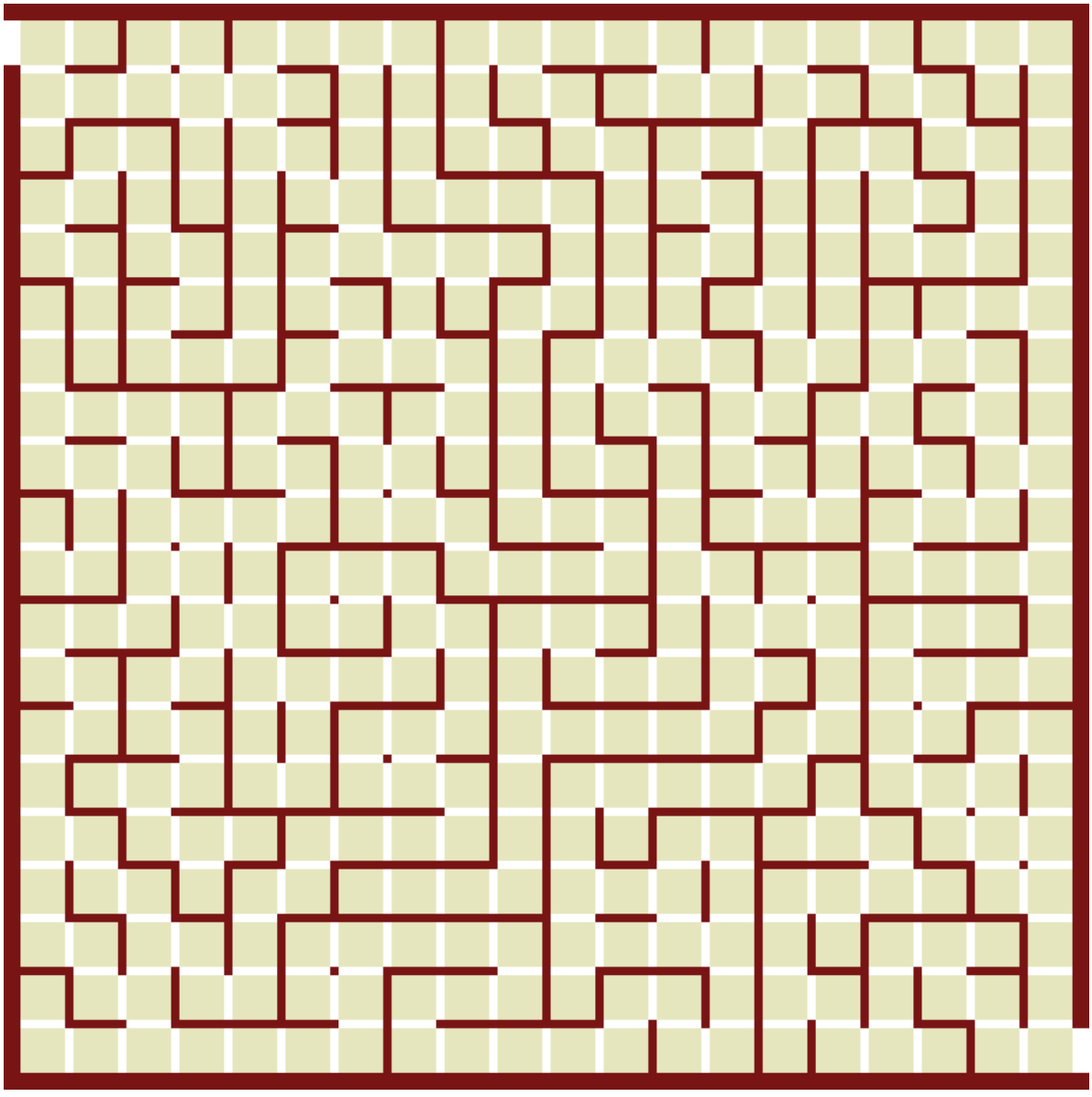
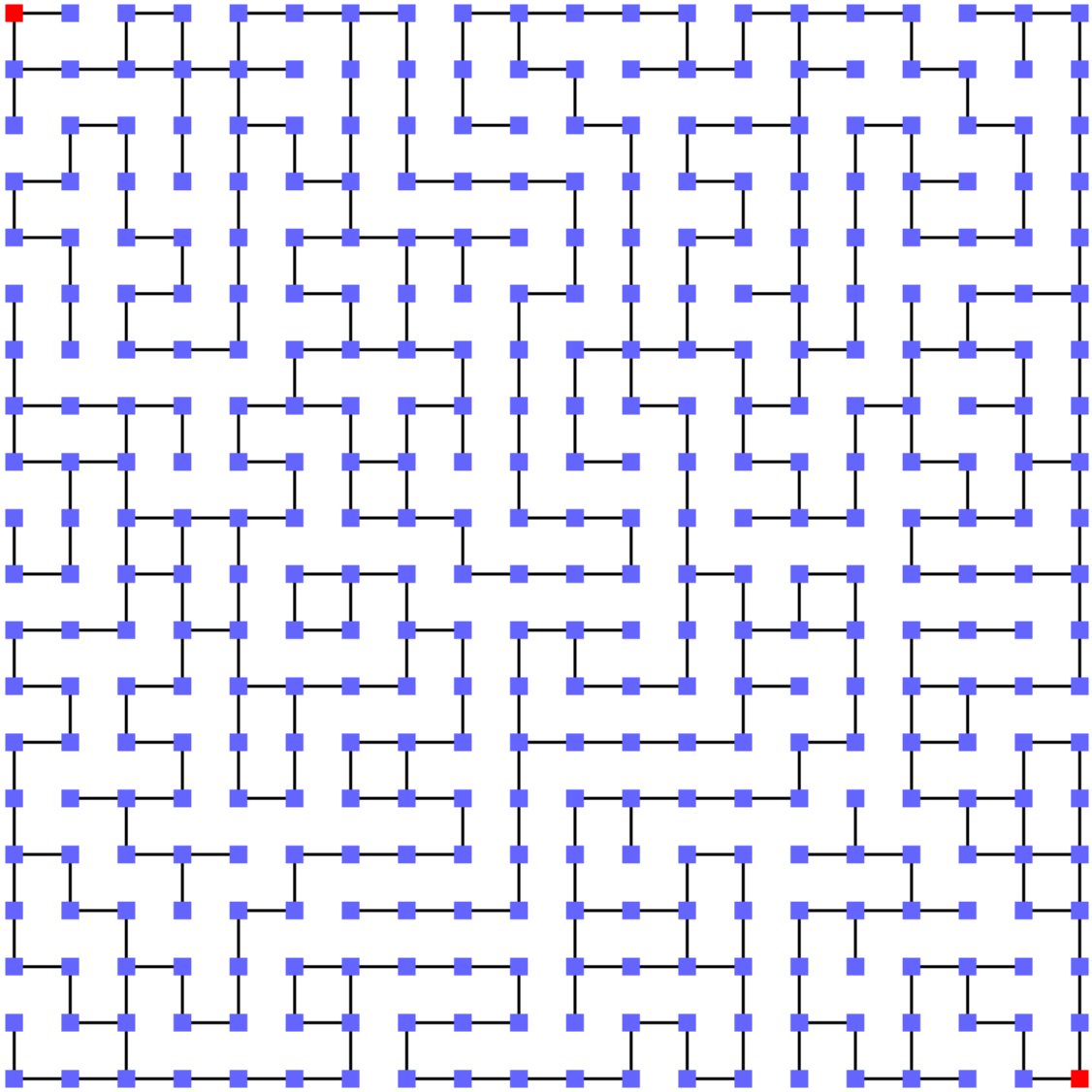
Progettazione di algoritmi

Discussione dell'esercizio [labirinto]

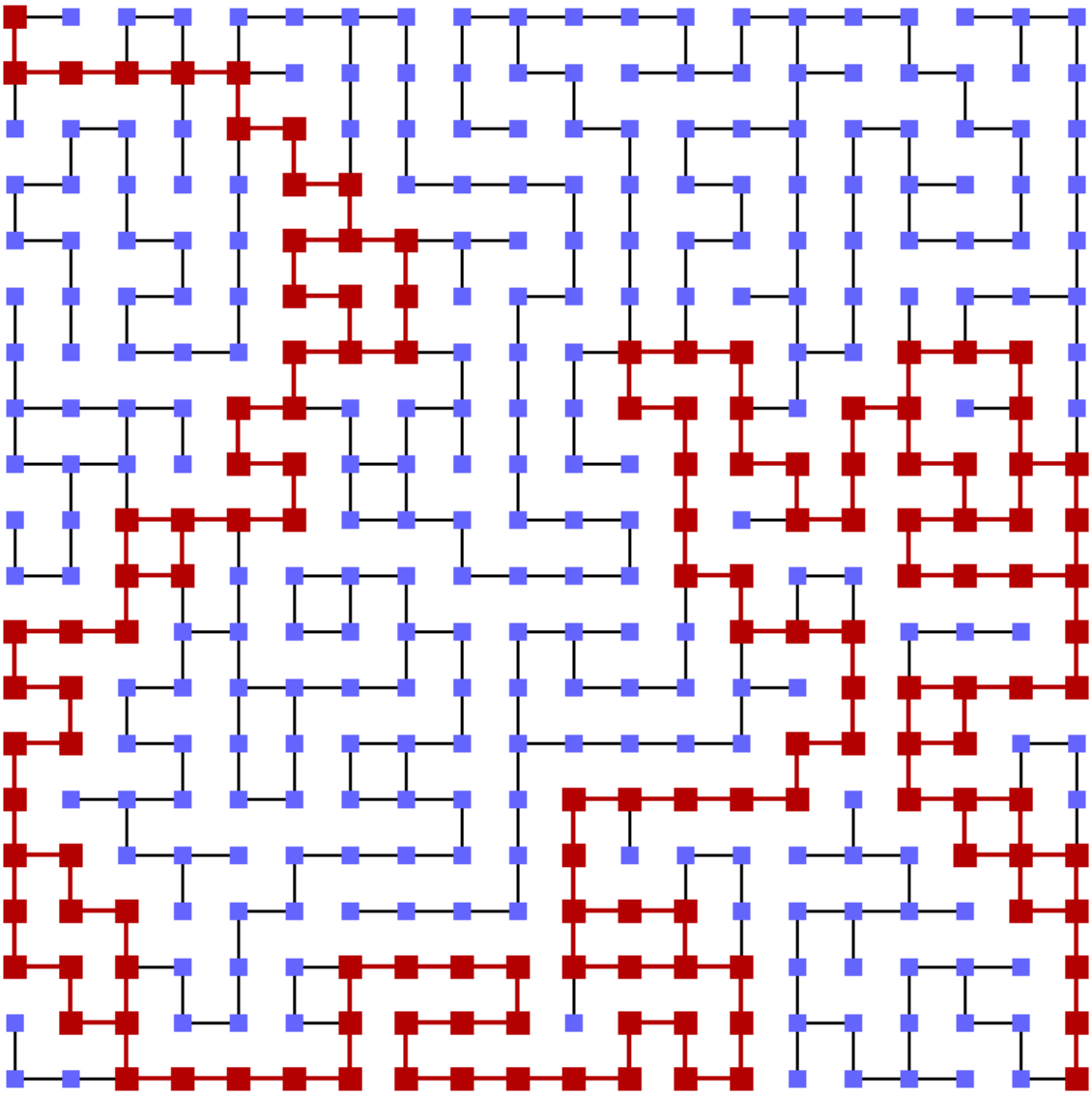
Nel testo dell'esercizio abbiamo considerato come lunghezza del percorso il numero di bivi ma possiamo stimare meglio la lunghezza reale del percorso se assumiamo che i corridoi (e i bivi) del labirinto siano lastricati con dei lastroni quadrati. Così possiamo rappresentare il labirinto tramite un grafo con un nodo per ogni lastrone e due nodi sono collegati da un arco (non diretto) se i corrispondenti lastroni sono adiacenti lungo un lato. La lunghezza di un cammino è così il numero di lastroni percorsi. La figura qui sotto mostra un esempio di labirinto con i lastroni di color sabbia e i muri in marrone.



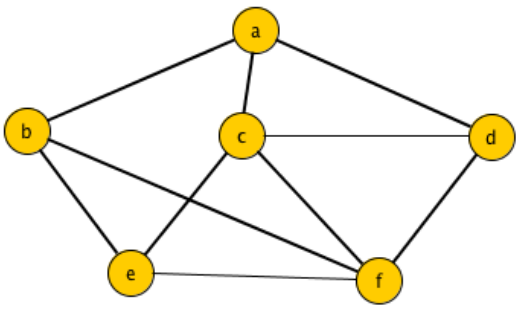
Il grafo corrispondente è il seguente, dove i nodi di entrata e uscita sono evidenziati in rosso:



L'esercizio chiede di dare un algoritmo che calcola il numero dei percorsi di lunghezza minima dall'entrata all'uscita. La figura seguente mostra tali cammini di lunghezza minima per il nostro labirinto. Ci sono esattamente 384 cammini di lunghezza minima 118. Nella figura tutti i nodi e gli archi che fanno parte dei 384 cammini sono evidenziati in rosso.



Come facciamo a calcolare il numero di cammini di lunghezza minima? Possiamo procedere come abbiamo già fatto per il calcolo delle distanze. Vogliamo calcolare il numero di cammini di lunghezza minima a partire da un dato nodo *u*. Un nodo *v* a distanza 1 può avere un solo cammino minimo. Ma un nodo a distanza 2 può avere più cammini minimi. Ad esempio, nel grafo qui sotto il nodo *e* ha due cammini minimi dal nodo *a* ((*a*,*b*,*e*) e (*a*,*c*,*e*)) e *f* ne ha tre.



In generale, per un nodo *w* a distanza *k* dal nodo di partenza *u* come possiamo calcolare il numero di cammini minimi? Un qualsiasi cammino di lunghezza *k* da *u* a *w* passa per un nodo *v* (il predecessore di *w*) che è a distanza *k* - 1 da *u*. Se il numero di cammini

minimi per v è $M(v)$, allora se ad ognuno di tali cammini aggiungiamo l'arco (v, w) otteniamo $M(v)$ cammini minimi per w. Questo ragionamento è valido per un qualsiasi nodo a distanza k - 1 da u che è adiacente a w. Quindi il numero di cammini minimi di w da u è uguale alla somma $M(v_1)+\ldots+M(v_h)$, dove $v_1,\ldots,v_h$ sono tutti i nodi a distanza k - 1 che sono adiacenti a w. Per implementare tale conteggio basterà modificare la BFS in modo tale da mantenere in un array `M` per ogni nodo visitato w il numero dei cammini minimi finora trovati `M[w]` e un array `Dist` per le distanze. Quando un nodo w è visitato per la prima volta tramite un nodo v, si porrà `M[w] <- M[v]` e `Dist[w] <- Dist[v] + 1`. Poi, ogni volta che si incontra un nodo v' adiacente a w e tale che `Dist[w] = Dist[v'] + 1`, si aggiorna il numero di cammini minimi di w, `M[w] <- M[w] + M[v']`. Ecco quindi lo pseudo-codice:

```
BFS_M(G: grafo, u: nodo)
    Dist: array delle distanze, inizializzato a -1
    M: array per il conteggio dei cammini minimi, inizializzato a 0
    Dist[u] <- 0
    M[u] <- 1
    Q <- coda vuota
    Q.enqueue(u)
    WHILE Q non è vuota DO
        v <- Q.dequeue()
        FOR ogni adiacente w di v DO
            IF Dist[w] = -1 THEN
                Dist[w] <- Dist[v] + 1
                M[w] <- M[v]
                Q.enqueue(w)
            ELSE IF Dist[w] = Dist[v] + 1 THEN
                M[w] <- M[w] + M[v]
    RETURN M
```

La complessità dell'algoritmo è pari a quella della BFS, cioè $O(n + m)$. Dopo aver ottenuto l'array `M` calcolato da `BFS_M(G, u)`, per ogni nodo v, `M[v]` dà il numero di cammini minimi da u a v.

Esercizi

Questa lezione è dedicata alla risoluzione di esercizi relativi ai cammini di lunghezza minima e alla BFS.

Esercizio [DFS vs BFS]

Quali caratteristiche deve avere un grafo non diretto e connesso perchè le visite DFS e BFS producano alberi di visita uguali, partendo dallo stesso nodo?

Esercizio [dall'albero alle distanze]

Dato un vettore dei padri `P` che rappresenta l'albero di una BFS a partire da un nodo u, dare un algoritmo che calcola il corrispondente array `Dist` delle distanze da u in tempo $O(n)$.

Esercizio [stessa distanza]

Descrivere un algoritmo efficiente che, dato un grafo non diretto e connesso e due suoi nodi u e v, trova i nodi che hanno la stessa distanza da u e v.

Esercizio [distanza tra insiemi]

Dato un grafo non diretto G e due sottoinsiemi A e B dei suoi nodi si definisce distanza tra A e B la distanza minima per andare da un nodo in A ad un nodo in B. Se A e B non sono disgiunti, la loro distanza è 0. Descrivere un algoritmo che, dato G e due sottoinsiemi dei nodi A e B calcola la loro distanza. L'algoritmo deve avere complessità $O(n + m)$.

Esercizio [Roma]

Descrivere un algoritmo che, dato un grafo diretto e fortemente connesso e un suo nodo v, trova tutti i cammini minimi tra tutte le coppie di nodi con la restrizione che questi cammini devono passare per v.

Esercizio per casa [acqua 2]

Ricordiamo il problema dei tre contenitori. Abbiamo tre contenitori di capienza 10, 7 e 4 litri, rispettivamente. Inizialmente, quelli da 4 e 7 litri sono pieni d'acqua e quello da 10 è vuoto. Possiamo fare un solo tipo di operazione: versare l'acqua da un contenitore in un altro fermandoci solo quando o il contenitore sorgente è vuoto o quello destinazione è pieno. Diciamo che una sequenza di operazioni di versamento è *buona* se termina lasciando esattamente 2 litri o nel contenitore da 7 o in quello da 4. Inoltre, diciamo che una sequenza buona è *parsimoniosa* se il totale dei litri versati in tutti i versamenti della sequenza è minimo rispetto a quello di tutte le sequenze buone. Descrivere un algoritmo per trovare una sequenza buona parsimoniosa.

Soluzioni

Di seguito presentiamo alcune possibili soluzioni degli esercizi proposti.

Discussione dell'esercizio [DFS vs BFS]

Sia G un grafo che, partendo da un nodo u, DFS e BFS producono la stesso albero di visita T radicato in u. Supponiamo che ci sia un arco $\{v, w\}$ che non appartiene a T. Sappiamo che nella DFS sarà classificato come arco all'indietro, cioè v è un antenato di w in T. Per ipotesi T è anche l'albero di visita della BFS e siccome v è un antenato di w e l'arco $\{v, w\}$ non appartiene a T, deve essere che $d(u, w) \geq d(u, v) + 2$. Ma questo non è possibile perché w è adiacente a v e quindi deve essere $d(u, w) \leq d(u, v) + 1$. Perciò concludiamo che un arco non appartenente a T non può esistere. Quindi i due alberi di visita sono uguali solo quando il grafo G è un albero.

Discussione dell'esercizio [dall'albero alle distanze]

Sia v un nodo qualsiasi, il calcolo della distanza da u tramite il vettore dei padri `P` può essere fatto come segue:

```
d <- 0
w <- v
WHILE P[w] <> w DO
    d <- d + 1
    w <- P[w]
OUTPUT d
```

È chiaro che questo richiede tempo lineare nella distanza d, cioè $O(d)$. Se lo ripetiamo per tutti i nodi avremo un algoritmo che richiede tempo lineare nella somma di tutte le distanze. In generale, tale algoritmo ha una complessità superiore a $O(n)$. Ad esempio, se il grafo è un cammino e quindi anche l'albero rappresentato da `P` è un cammino, la complessità è $O(n^2)$.

Ma se nell'effettuare il calcolo delle distanze non ricalcoliamo le distanze che abbiamo già calcolato, potremmo risparmiare parecchio tempo.

```
DISTANZE(P: vettore dei padri)
    Dist: array delle distanze, inizializzato a -1
    FOR ogni nodo w DO
        IF Dist[w] = -1 THEN
            Dist[w] <- DIST(P, w, Dist)
    RETURN Dist

DIST(P: vettore dei padri, w: nodo, Dist: array)
    IF Dist[w] = -1 THEN
        IF P[w] = w THEN
            Dist[w] <- 0
        ELSE
            Dist[w] <- DIST(P, P[w], Dist) + 1
    RETURN Dist[w]
```

La funzione `DIST(P, w, Dist)` calcola la distanza di w solo se non è già stata calcolata e sfrutta le distanze già calcolate. La chiamata ricorsiva è effettuata solamente se la distanza di `w` non è già stata calcolata. Quindi il numero totale di chiamate ricorsive effettuate in tutte le chiamate a `DIST` della funzione `DISTANZE` non può superare il numero di nodi. Questo proprio perché per ogni nodo sarà effettuata la chiamata ricorsiva solo la prima volta, quando la sua distanza non è stata ancora calcolata. Siccome ogni chiamata a `DIST` costa tempo costante, escluso il tempo preso dall'eventuale chiamata ricorsiva, l'algoritmo ha complessità lineare nel numero di nodi, cioè $O(n)$.

Discussione dell'esercizio [stessa distanza]

Per determinare i nodi a uguale distanza dai nodi u e v basterà fare una BFS da u e una BFS da v per ottenere gli array delle distanze `Dist_u` da u e quello delle distanze da v `Dist_v`. Poi per ogni nodo w, si controlla se `Dist_u[w] = Dist_v[w]`.

```
SameDist(G: grafo, u, v: nodi)
    Dist_u <- BFS(G, u)
    Dist_v <- BFS(G, v)
    S <- lista vuota
    FOR ogni nodo w DO
        IF Dist_u[w] = Dist_v[w] THEN
            S.append(w)
    RETURN S
```

Chiaramente questo algoritmo ha complessità O($n + m$).

Discussione dell'esercizio [distanza tra insiemi]

Per determinare le distanze da un insieme di nodi A, piuttosto che da un nodo solo, si può pensare ad una naturale estensione della BFS. Invece di partire da un singolo nodo, la BFS inizia da tutti i nodi dell'insieme A simultaneamente. Così ogni nodo in A avrà la distanza inizializzata a 0. Poi si considera ogni adiacente v di questi ultimi, se v non è già stato visitato, la sua distanza sarà 1. Poi si considerano gli adiacenti dei nodi a distanza 1, quelli non ancora visitati avranno distanza 2, e così via, proprio come nella normale BFS.

```
BFS_SET(G: grafo, A: insieme di nodi)
    Dist_A <- array delle distanze dall'insieme A, inizializzato a -1
    Q <- coda vuota
    FOR ogni nodo u in A DO
        Dist_A[u] <- 0
        Q.enqueue(u)
    WHILE Q non è vuota DO
        v <- Q.dequeue()
        FOR ogni adiacente w di v DO
            IF Dist_A[w] = -1 THEN
                Dist_A[w] = Dist_A[v] + 1
                Q.enqueue(w)
    RETURN Dist_A
```

Si osservi che l'inizializzazione di `Dist_A` e di `Q` prendono tempo al più lineare in n e poi la visita procede come una normale BFS (anche se parte da un insieme di nodi invece che da uno solo). Quindi la complessità è sempre O($n + m$). Una volta ottenuto l'array delle distanze da A è facile determinare la distanza tra A e B:

```
DIST(G: grafo, A, B: insiemi di nodi)
    Dist_A <- BFS_SET(G, A)
    d <- -1
    FOR ogni nodo u in B DO
        IF d = -1 OR Dist_A[u] < d THEN
            d = Dist_A[u]
    RETURN d
```

Chiaramente, anche l'ultima parte dell'algoritmo è efficiente avendo complessità O(n). Quindi l'algoritmo ha complessità O($n + m$).

Discussione dell'esercizio [Roma]

Dati due nodi qualsiasi u e w dobbiamo trovare il cammino di lunghezza minima da u a w che però passa per il nodo v. Tale cammino sarà formato da due parti: un cammino da u a v e un cammino da v a w. Ovviamente, entrambi questi cammini sono di lunghezza minima. Si noti però che il cammino totale, da u a w, non è necessariamente semplice, cioè può passare due volte per lo stesso nodo. Quindi, per trovare questo cammino minimo vincolato a passare per v, possiamo trovare un cammino minimo c_u da v a u nel grafo con gli archi invertiti (cioè, il grafo trasposto) e poi un cammino minimo c_w da v a w (nel grafo originale). La concatenazione di c_u e c_w ci dà il cammino minimo da u a w che passa per v.

Allora, possiamo calcolarci come prima cosa gli alberi dei cammini minimi di G da v e quello dei cammini minimi di GT da v, dove GT è il grafo trasposto di G.

```
CENTER(G: grafo diretto, v: nodo)
    P <- BFS(G, v)     /* vettore dei padri dell'albero dei cammini minimi di G da v */
    GT <- TRASP(G)     /* ritorna il grafo trasposto */
    PT <- BFS(GT, v)   /* vettore dei padri dell'albero dei cammini minimi di GT da v */
    RETURN P, PT
```

La complessità dell'algoritmo è O($n + m$) dato che le due BFS prendono tempo O($n + m$) e anche la procedura per il calcolo del grafo trasposto prende tempo O($n + m$). Poi dati due qualsiasi nodi u e w possiamo costruirci il cammino minimo da u a w che passa per v nel seguente modo:

```
C_u <- lista vuota
DO
    C.append(u)
    u <- PT[u]
WHILE u <> v
C_w <- lista vuota
WHILE w <> v DO
    C_w.head(w)
    w <- P[w]
C <- concatenazione di C_u e C_w
OUTPUT C
```

Quindi una volta calcolati i vettori dei padri `P` e `PT` , un cammino minimo da u a v vincolato a passare per v si può ottenere in tempo lineare nella lunghezza del cammino.