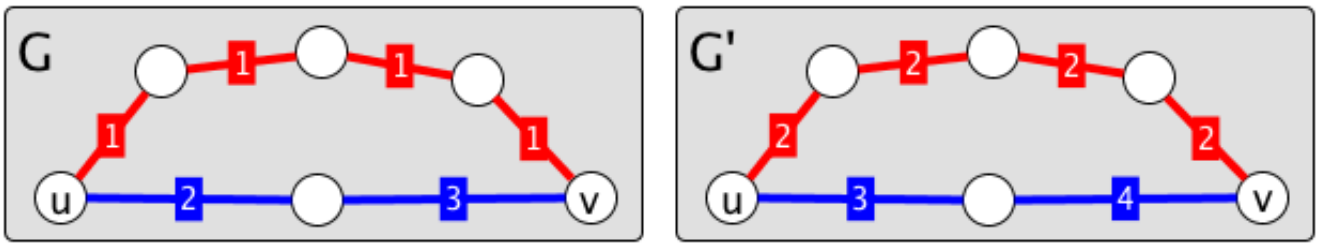
Progettazione di Algoritmi - lezione 10

Discussione dell'esercizio [pesi]

Nel grafo G' il peso di ogni arco (u, v) è $p'(u, v) = p(u, v) + 1$, dove $p(u, v)$ è il peso dell'arco nel grafo G. Il peso di un cammino C in G' è quindi $p'(C) = p(C) + L(C)$, dove $L(C)$ è la lunghezza di C e $p(C)$ è il peso di C in G. Si intuisce allora che potrebbe esistere un cammino minimo A in G tra due nodi u e v tale che non è più un cammino minimo in G', perché il peso di A in G' è $p'(A) = p(A) + L(A)$ e potrebbe esserci un cammino B tra u e v tale che $p(B) > p(A)$ (cioè B non è un cammino minimo in G), $L(B) < L(A)$ e $p'(B) < p'(A)$ (cioè, B è un cammino minimo in G' e A non lo è). Nella figura qui sotto è mostrato proprio un esempio di una tale situazione:

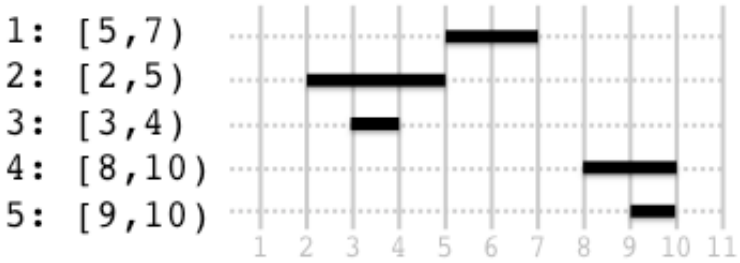


Il cammino in rosso è un cammino minimo di peso 4 in G tra u e v ma non lo è più in G' perchè in G' ha peso 8 e il cammino in blu ha peso 7. Quindi l'albero dei cammini minimi da u in G è diverso da quello in G'.

Passiamo ora alla seconda domanda dell'esercizio. Supponiamo che G'' è ottenuto raddoppiando i pesi degli archi in G. Un qualsiasi cammino C che in G ha peso $p(C)$ avrà peso $p''(C) = 2p(C)$ in G''. Ne segue che, se A e B sono due qualsiasi cammini tali che $p(A) \leq p(B)$, risulta $p''(A) = 2p(A) \leq 2p(B) = p''(B)$. Quindi le relazioni d'ordine tra i pesi dei cammini in G'' rimangono le stesse di quelle in G. Dunque anche i cammini minimi e gli alberi dei cammini minimi rimagono gli stessi.

Selezione attività

Per illustrare il progetto e l'analisi di algoritmi greedy consideriamo un problema piuttosto semplice chiamato *Selezione Attività*: date n attività (lezioni, seminari ecc.) e pr ogni attività i, $1 \leq i \leq n$, l'intervallo temporale $[s_i, f_i)$ in cui l'attività dovrebbe svolgersi, selezionare il maggior numero di attività che possono essere svolte senza sovrapposizioni in un'aula. Ad esempio, se si hanno le seguenti 5 attività:



se ne possono selezionare 3 (ad esempio, le attività 1,2,4 o 1,3,5).

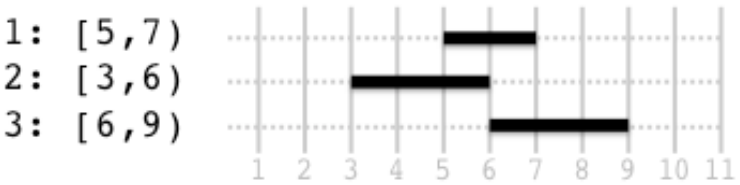
È naturale considerare come soluzione parziale un qualsiasi insieme di attività che non si sovrappongono. Considerare invece come soluzione parziale un insieme di attività che possono anche sovrapporsi non è in linea con la filosofia degli algoritmi greedy. Infatti, una soluzione parziale dovrebbe essere sempre estesa ma mai ridotta. Vale a dire, non si torna mai su una decisione che è stata già presa. Anche il concetto di estensione locale è naturale, semplicemente, l'aggiunta di una attività che non si sovrappone a quelle già presenti nella soluzione parziale. Invece, il criterio per misurare la convenienza di una estensione locale non è così facile da scegliere. Un possibile criterio potrebbe consistere nel preferire, tra tutte le attività che possiamo aggiungere, quella (o una di quelle) che inizia prima. Oppure quella che dura meno; oppure quella che termina prima. Quindi potremmo proporre almeno i seguenti tre algoritmi greedy per il problema Selezione Attività:

```
INIZIA_PRIMA([s1, f1), [s2, f2),... [sn, fn): intervalli delle n attività)
    SOL <- insieme vuoto
    WHILE esistono attività che si possono aggiungere a SOL senza sovrapposizioni DO
        Sia k un'attività con minimo tempo d'inizio fra quelle che si possono aggiungere a SOL
        SOL <- SOL ∪ {k}
    OUTPUT SOL

DURA_MENO([s1, f1), [s2, f2),... [sn, fn): intervalli delle n attività)
    SOL <- insieme vuoto
    WHILE esistono attività che si possono aggiungere a SOL senza sovrapposizioni DO
        Sia k un'attività di minima durata fra quelle che si possono aggiungere a SOL
        SOL <- SOL ∪ {k}
    OUTPUT SOL

TERMINA_PRIMA([s1, f1), [s2, f2),... [sn, fn): intervalli delle n attività)
    SOL <- insieme vuoto
    WHILE esistono attività che possono essere aggiunte a SOL senza sovrapposizioni DO
        Sia k un'attività con minimo tempo di fine fra quelle che si possono aggiungere a SOL
        SOL <- SOL ∪ {k}
    OUTPUT SOL
```

Ora vogliamo analizzare i tre algoritmi. Prima di avventurarsi in un tentativo di dimostrazione di correttezza di un algoritmo conviene pensare a qualche semplice istanza del problema che potrebbe mettere in difficoltà l'algoritmo. Nell'algoritmo INIZIA_PRIMA le attività sono scelte in ordine di tempo d'inizio. Quindi se c'è un'attività che inizia prima di altre con le quali è incompatibile mentre queste sono tra loro compatibili allora la scelta di tale attività può pregiudicare il raggiungimento di una soluzione ottima. Infatti, consideriamo la seguente semplice istanza:

Chiaramente, l'algoritmo fornirebbe la soluzione composta dalla sola attività 1 mentre la soluzione ottima ha due attività (2 e 3). Quindi l'algoritmo non è corretto.

Nell'algoritmo DURA_MENO le attività sono scelte in ordine di durata. Se applicato all'istanza precedente in effetti l'algoritmo trova la soluzione ottima. Però potrebbe esserci un'attività con una durata piccola che è incompatibile con due attività di durata maggiore che sono tra loro compatibili. La scelta dell'attività piccola può pregiudicare il raggiungimento dell'ottimo. Infatti, consideriamo l'istanza:



Chiaramente, l'algoritmo fornisce la soluzione composta dalla sola attività 1 mentre la soluzione ottima ha due attività (2 e 3). Quindi l'algoritmo non è corretto.

L'algoritmo TERMINA_PRIMA produce soluzioni ottime per entrambe le istanze che mettono in difficoltà gli altri due algoritmi. È corretto? Se non si riesce a trovare un'istanza che mette in difficoltà un algoritmo, cioè, non si riesce a trovare un controesempio alla correttezza dell'algoritmo, allora si può tentare di dimostrarne la correttezza. Il tentativo potrebbe fallire e il modo in cui fallisce potrebbe darci indicazioni su come trovare un controesempio. Ma a volte il tentativo fallisce nonostante l'algoritmo sia corretto. Semplicemente, potrebbe essere assai arduo trovare una dimostrazione di correttezza.

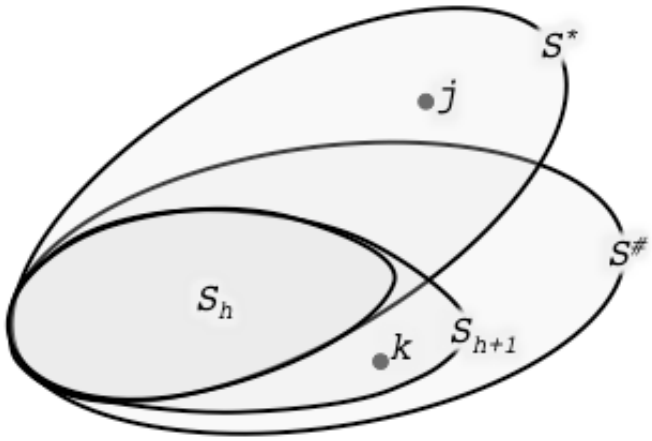
Schema generale per dimostrare la correttezza di algoritmi greedy (se sono corretti)

Non ci sono ricette o procedure meccaniche che permettano di costruire dimostrazioni. Spesso costruire una dimostrazione (non banale) richiede ingegnosità, creatività e fantasia. Le dimostrazioni di correttezza di algoritmi non fanno eccezione. Però nel caso di algoritmi greedy si possono dare delle linee guida che spesso risultano molto utili nella costruzione di una dimostrazione di correttezza. Queste forniscono uno schema generale che decompone l'ipotetica dimostrazione in parti abbastanza piccole e ben delineate così da facilitarne l'analisi. Prima di tutto si cerca di dimostrare che ogni soluzione parziale prodotta ad un certo passo dell'algoritmo è estendibile ad una opportuna soluzione ottima. Ciò equivale a dimostrare che le decisioni prese dall'algoritmo non pregiudicano mai l'ottenimento di una soluzione ottima (non prende mai decisioni "sbagliate"). Dopo aver dimostrato ciò si sa per certo che la soluzione finale prodotta dall'algoritmo è estendibile ad una soluzione ottima. Rimane quindi da dimostrare che la soluzione finale è invero uguale a tale soluzione ottima. Ciò equivale a dimostrare che l'algoritmo prende un numero sufficiente di decisioni "giuste". Altri aspetti dello schema generale saranno evidenziati dopo averlo visto applicato all'algoritmo TERMINA_PRIMA.

Preliminarmente è necessario dimostrare che l'algoritmo effettua sempre un numero finito di passi. Nel caso dell'algoritmo TERMINA_PRIMA è sufficiente osservare che ad ogni iterazione del `WHILE` viene aggiunta una nuova attività a `SOL` e la condizione del `WHILE` diventa falsa non appena non ci sono più attività che possono essere aggiunte a `SOL`. Quindi il `WHILE` può eseguire al più n iterazioni. Indichiamo con SOL_h la soluzione parziale prodotta durante l'h-esima iterazione del `WHILE` e sia SOL_0 la soluzione parziale iniziale. Sia m il numero di iterazioni eseguite dal `WHILE`. Nel caso dell'algoritmo TERMINA_PRIMA dimostrare che ogni soluzione parziale prodotta è estendibile ad una soluzione ottima significa dimostrare la seguente affermazione:

Per ogni $h = 0,1,\ldots m$ esiste una soluzione ottima SOL^* che contiene SOL_h.

Dimostrazione La dimostrazione procede per induzione su h. Per $h = 0$ l'asserto è banalmente vero in quanto SOL_0 è vuoto. Sia ora $h \geq 0$, vogliamo dimostrare che se l'asserto è vero per h allora è anche vero per $h + 1$ (sempreché $h + 1 \leq m$). L'ipotesi induttiva ci dice quindi che esiste una soluzione ottima SOL^* che contiene SOL_h. Se SOL^* contiene anche SOL_{h+1}, abbiamo fatto. Se invece SOL^* non contiene SOL_{h+1}, l'attività k aggiunta a SOL_h nell'$(h + 1)$-esima iterazione non può appartiene a SOL^*. Vogliamo far vedere che è possibile trasformare SOL^* in un'altra soluzione ottima che invece contiene SOL_{h+1}. Tale soluzione ottima dovrà contenere l'attività k (oltre a tutte le attività di SOL_h) e siccome non può superare la cardinalità di SOL^*, non dovrà contenere qualche attività che è in SOL^*. Dobbiamo trovare un'attività in SOL^* che può essere sostituita dall'attività k. Osserviamo che SOL_{h+1} è una soluzione ammissibile, cioè, le attività in SOL_{h+1} sono tutte compatibili fra loro. Infatti SOL_h è ammissibile perché è contenuta in una soluzione ottima e l'attività k è una attività che può essere aggiunta a SOL_h, cioè, è compatibile con tutte le attività in SOL_h. Quindi, $|SOL^*| \geq |SOL_{h+1}|$ e allora c'è almeno un'attività in SOL^* che non appartiene a SOL_{h+1}. Sia j quella fra queste attività che termina per prima. Definiamo $SOL^{\#} = (SOL^* - \{j\}) \cup \{k\}$ e mostriamo che $SOL^{\#}$ è una soluzione ammissibile.



Siccome j appartiene a SOL^* e non appartiene a SOL_h, allora j era fra le attività che potevano essere aggiunte a SOL_h nell'$(h + 1)$-esima iterazione. Dato che è stata scelta l'attività k, deve essere $f_k \leq f_j$. Allora non ci sono altre attività in SOL^* oltre a j che sono incompatibili con k. Infatti, tutte le attività in $(SOL^* - \{j\}) - SOL_{h+1}$ terminano dopo j e sono compatibili con j, per cui iniziano non prima di f_j e quindi non prima di f_k. Dunque $SOL^\#$ è una soluzione ammissibile con la stessa cardinalità di una soluzione ottima, quindi è una soluzione ottima. Inoltre, $SOL^\#$ contiene SOL_{h+1} e questo completa la dimostrazione.

A questo punto sappiamo che la soluzione finale SOL_m prodotta dall'algoritmo è contenuta in una soluzione ottima SOL^*. Supponiamo per assurdo che SOL_m è differente da SOL^*. Allora, esiste almeno un'attività j in SOL^* che non appartiene a SOL_m. Siccome l'attività j è compatibile con tutte le attività in SOL_m e non appartiene a SOL_m la condizione del `WHILE` all'inizio della $(m + 1)$-esima iterazione è vera in contraddizione con il fatto che il `WHILE` esegue solamente m iterazioni. Dunque deve essere $SOL_m = SOL^*$. Abbiamo così dimostrato che l'algoritmo TERMINA_PRIMA è corretto per il problema Selezione Attività.

Le dimostrazioni che seguono lo schema generale condividono alcuni altri aspetti oltre a quelli già menzionati. La dimostrazione che ogni soluzione parziale è estendibile a una soluzione ottima può sempre essere fatta per induzione sulle iterazioni dell'algoritmo. Anzi, l'enunciato da dimostrare ha proprio quella forma per facilitare una dimostrazione per induzione. Ciò che interessa, come si vede dalla dimostrazione precedente, è dimostrare che la soluzione finale prodotta dall'algoritmo è estendibile ad una soluzione ottima. Un altro aspetto che tali dimostrazioni condividono è che nel corso della dimostrazione per induzione occorre trasformare la soluzione ottima, che per ipotesi induttiva estende la soluzione parziale di una certa iterazione, in un'altra soluzione ottima che estende la soluzione parziale dell'iterazione successiva. Spesso tale trasformazione consiste in una sostituzione di un opportuno elemento della soluzione ottima con un elemento della soluzione parziale. Questo è l'aspetto cruciale perché è soprattutto nella determinazione di tale trasformazione che le caratteristiche specifiche dell'algoritmo entrano in gioco.

Implementazione dell'Algoritmo TERMINA_PRIMA

Per ragionare circa la correttezza di un algoritmo conviene che la sua descrizione sia la più semplice possibile, libera cioè da tutti quei dettagli che servono a specificare un'implementazione ma che non influenzano le scelte che l'algoritmo effettua. Una volta che la correttezza di un algoritmo è stata dimostrata si può passare a determinare quei dettagli che permettono di specificare un'implementazione efficiente.

Tornando all'algoritmo TERMINA_PRIMA possiamo supporre innanzitutto che l'input sia rappresentato da un array i cui elementi sono le coppie tempo d'inizio, tempo di fine delle attività. Ad ogni iterazione occorre scegliere l'attività con il minimo tempo di fine tra quelle compatibili. Per rendere efficiente questa scelta conviene ordinare preliminarmente le attività per tempi di fine non decrescenti. Fatto questo, per determinare la compatibilità è

sufficiente controllare che il tempo d'inizio della prossima attività non preceda il tempo di fine dell'ultima attività selezionata. La descrizione di questa implementazione è la seguente:

```
TERMINA_PRIMA(A: array tale che A[i].s e A[i].f sono i tempi d'inizio e di fine dell'attività i
    SOL <- lista vuota
    Calcola un array P che ordina l'array A rispetto ai tempi di fine, cioè,
    A[P[1]].f ≤ A[P[2]].f ≤ … ≤ A[P[n]].f
    t <- 0           /* Assumiamo che i tempi d'inizio sono non negativi */
    FOR j = 1 TO n DO
        IF A[P[j]].s ≥ t THEN
            SOL.append(P[j])
            t <- A[P[j]].f
    OUTPUT SOL
```

L'ordinamento si può fare in O(*n*log*n*) usando uno degli algoritmi classici (ad esempio merge-sort). Ciascuna delle *n* iterazioni del `FOR` ha costo costante. Quindi, questa implementazione dell'algoritmo TERMINA_PRIMA ha complessità asintotica O(*n*log*n*), dove *n* è il numero di attività.

Correttezza dell'algoritmo di Dijkstra

Dimostreremo la correttezza dell'algoritmo di Dijkstra seguendo lo schema generale per gli algoritmi greedy. Ricordiamo l'algoritmo:

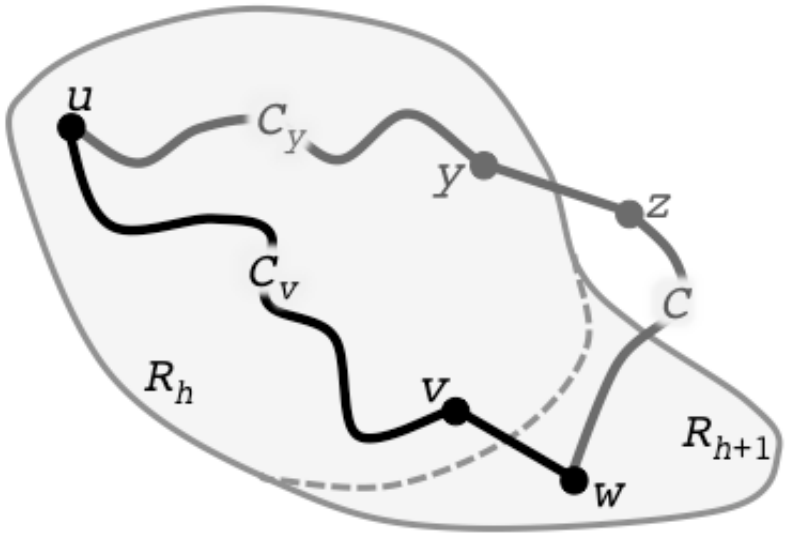
```
DIJKSTRA(G: grafo, u: nodo)
    Dist: array delle distanze, inizializzato a infinito
    Dist[u] <- 0
    R <- {u}          /* Insieme dei nodi la cui distanza è già stata determinata */
    WHILE esiste un arco da R a fuori di R DO
        Trova un arco (v, w) con v in R, w non in R per cui è minima Dist[v] + p(v, w)
        Dist[w] <- Dist[v] + p(v, w)
        R <- R ∪ {w}
    RETURN Dist
```

Per prima cosa sinceriamoci che l'algoritmo termina sempre. Se la condizione del `WHILE` è vera, allora un nuovo arco è scelto e un nuovo nodo è aggiunto ad `R`. Ne segue che il `WHILE` non può eseguire più di *n*–1 iterazioni. Una soluzione parziale è rappresentata da un array parziale delle distanze, cioè, un array definito solamente in alcuni elementi. L'insieme in cui è definito è dato proprio da `R`. Così, denotiamo con $Dist_h$ ed R_h rispettivamente l'array `Dist` e l'insieme `R` al termine della *h*-esima iterazione. Come al solito $Dist_0$ ed R_0 denotano i valori iniziali. Sia *k* il numero di iterazioni eseguite. Dobbiamo dimostrare che $Dist_h$ è estendibile ad una soluzione ottima. In questo caso c'è un'unica soluzione ottima che è $d_G(u, \cdot)$, cioè le distanze dal nodo *u*.

> **Per ogni $h = 0,1,\ldots,k$, $Dist_h$ è uguale a $d_G(u, \cdot)$ su R_h, cioè, per ogni x in R_h, $Dist_h[x] = d_G(u, x)$.**

Dimostrazione Procediamo per induzione su h. Per $h = 0$, è vero perchè $R_0 = \{u\}$ e $Dist_0[u] = 0 = d_G(u, u)$. Sia ora $0 \le h < k$. Per ipotesi induttiva, per ogni x in R_h, $Dist_h[x] = d_G(u, x)$. Vogliamo allora mostrare che la tesi è vera anche per $h + 1$. Sia (v, w) l'arco scelto nella $(h + 1)$-esima iterazione, per cui $R_{h+1} = R_h \cup \{w\}$. Dobbiamo quindi dimostrare che $Dist_{h+1}[w] = d_G(u, w)$. Per l'ipotesi induttiva $Dist_h[v] = d_G(u, v)$, perciò esiste un cammino C_v da u a v di peso $Dist_{h+1}[v]$. Sia C_w il cammino da u a w che si ottiene concatenando C_v con l'arco (v, w). Siccome $Dist_{h+1}[w] = Dist_{h+1}[v] + p(v, w)$, il peso di

C_w è proprio pari a $Dist_{h+1}[w]$. Quindi deve essere $Dist_{h+1}[w] \geq d_G(u, w)$. Rimane da far vedere la disuguaglianza opposta. Sia C un cammino di peso minimo da u a w, cioè, $p(C) = d_G(u, w)$. Percorrendo il cammino C da u verso w, sia z il primo nodo che s'incontra che non appartiene a R_h. Siamo sicuri che z esiste perché almeno l'ultimo nodo di C, cioè w, non appartiene a R_h. Sia y il predecessore di z in C. Siamo sicuri che y esiste perchè C inizia in u e siccome u appartiene a R_h non può essere che z sia proprio u. Tutto ciò implica che y è in R_h, z non è in R_h ed esiste l'arco (y, z). Quindi l'arco (y, z) era uno degli archi che potevano essere scelti nella $(h + 1)$-esima iterazione. Siccome è stato scelto (v, w), deve essere $Dist_h[v] + p(v, w) \leq Dist_h[y] + p(y, z)$. Sia C_y la parte del cammino C che va da u a y.



Chiaramente $p(C_y) = d_G(u, y)$ e per l'ipotesi induttiva $Dist_h[y] = d_G(u, y)$. Inoltre, $p(C) \geq p(C_y) + p(y, z)$, dato che il peso degli eventuali altri archi di C è non negativo. Mettendo insieme quanto finora dimostrato si ottiene: $d_G(u, w) = p(C) \geq p(C_y) + p(y, z) = Dist_h[y] + p(y, z) \geq Dist_h[v] + p(v, w) = Dist_{h+1}[w]$, e la dimostrazione è completa.

Una volta dimostrato che l'algoritmo calcola in modo corretto le distanze da u, risulta chiaro che l'albero dei cammini costruito e proprio un albero dei cammini minimi da u. Infatti, gli archi aggiunti all'albero sono precisamente gli archi che determinano le distanze. La dimostrazione formale ricalca quella data per le distanze ed è lasciata come esercizio.

Esercizio [rifornimenti]

Si supponga di dover effettuare un viaggio dalla località A alla località B con un'auto che ha un'autonomia di k chilometri. Lungo la strada ci sono $n + 1$ distributori di benzina ciascuno distante dal precedente meno di k chilometri. Sia d_i la distanza che separa il distributore i dal distributore $i + 1$ per $i = 1,2,\ldots n$, dove il distributore 1 è in A e il distributore $n + 1$ è in B.

Inizialmente il serbatoio dell'auto è vuoto. Descrivere un algoritmo greedy che preso in input la lista delle distanze $d_1,\ldots d_n$ dei distributori, seleziona un numero minimo di distributori in cui far rifornimento durante il viaggio. Provare la correttezza dell'algoritmo e valutare la complessità di un'implementazione efficiente.