Spiegare la realizzazione di grafi mediante matrice di adiacenza e matrice di incidenza, fornendo vantaggi e svantaggi di ognuna:

COS'E' UNA MATRICE DI INCIDENZA?

Le più semplici rappresentazioni di grafi utilizzano matrici. Tra queste le più usate sono la matrice di incidenza nodi-archi e la matrice di adiacenza nodi-nodi.

La matrice di incidenza nodi-archi è una matrice rettangolare $B = [b_{ik}]$ di dimensioni $n \times m$, dove ciascuna riga rappresenta un nodo e ciascuna colonna rappresenta un arco, tale che:

$$b_{ik} = \begin{cases} -1, & \text{se l'arco } k\text{-esimo esce dal nodo } i, \\ +1, & \text{se l'arco } k\text{-esimo entra nel nodo } i, \\ 0, & \text{altrimenti.} \end{cases}$$

Se il grafo è non orientato, invece, si ha:

$$b_{ik} = \begin{cases} 1, & \text{se l'arco } k\text{-esimo è incidente nel nodo } i, \\ 0, & \text{altrimenti.} \end{cases}$$

VANTAGGI E SVANTAGGI MATRICE INCIDENZA:

La matrice di incidenza è molto utile per rappresentare un grafo in certi problemi di PROGRAMMAZIONE LINEARE, nei quali si cerca un vettore x di m incognite (reali o intere) associate agli archi che soddisfi un sistema di equazioni del tipo $Bx = c$, dove c è un vettore di n termini noti interi. Purtroppo, con questa rappresentazione, lo spazio di memoria occupato è sempre $\Theta(nm)$. Inoltre, dato un nodo, non è agevole ricavare l'insieme di adiacenza. Infatti per calcolare un qualsiasi $A(u)$ occorre scandire la riga u di B alla ricerca delle colonne k con elementi $b_{uk} = -1$ e per ciascuna colonna k scandirla per individuare l'indice di riga v tale che $b_{vk} = +1$, per un totale di $\Theta(nm)$ tempo.

COS'E' UNA MATRICE DI ADIACENZA?

La matrice di adiacenza nodi-nodi, invece, è una matrice quadrata $E = [e_{ij}]$ di dimensioni $n \times n$, tale che:

$$e_{ij} = \begin{cases} 1, & \text{se } (i,j) \in A, \\ 0, & \text{se } (i,j) \notin A. \end{cases}$$

Se il grafo è non orientato, allora tale matrice è simmetrica, ovvero $e_{ij} = e_{ji}$ per ogni i e j.

VANTAGGI E SVANTAGGI MATRICE ADIACENZA:

Con questa realizzazione è banale verificare in tempo $O(1)$ se un dato arco è presente nel grafo oppure no. Purtroppo, il tempo per ricavare l'insieme di adiacenza $A(u)$ di un qualsiasi nodo u è sempre $\Theta(n)$, indipendentemente dalla cardinalità di $A(u)$, perché occorre scandire un'intera riga della matrice alla ricerca degli elementi $b_{uv} = 1$. Inoltre, lo spazio di memoria occupato è $\Theta(n^2)$ e il tempo per esaminare tutti gli archi è $\Theta(n^2)$, anche se il grafo è sparso, cioè contiene un numero m di archi che è $O(n)$. Se il grafo è pesato, allora nella matrice E si utilizzano i pesi degli archi al posto degli elementi binari. Se p_{ij} è il peso dell'arco (i, j), allora la matrice E diventa:

$$e_{ij} = \begin{cases} p_{ij}, & \text{se } (i,j) \in A, \\ +\infty \text{ (oppure } -\infty) & \text{se } (i,j) \notin A. \end{cases}$$

Strutture dati per la sua implementazione

Per “visitare” almeno una volta ogni nodo ed ogni arco di un grafo orientato fortemente connesso (o non orientato e connesso). Il metodo effettua la visita in tempo ottimo ® (n + m) a partire da un generico nodo r:

2. Breadth-First-Search (BFS), ovvero visita in ampiezza (detta anche a ventaglio).

Nella visita BFS, invece, i nodi sono visitati in ordine di distanza crescente dal nodo di partenza r, dove la distanza da r ad un generico nodo u è il minimo numero di archi in un cammino (o catena) da r ad u. Tale metodo di visita è una estensione di una visita per livelli di un albero radicato, in cui i figli di un nodo sono visitati dopo aver visitato tutti gli altri nodi che stanno allo stesso livello del padre.

In entrambi i metodi, occorre mantenere una lista dei nodi che sono già stati visitati, ma i cui nodi adiacenti non sono ancora stati visitati. Poiché la DFS torna indietro dal più recente vicolo cieco, tale lista è di fatto una pila. D'altro lato, poiché la BFS visita i nodi più vicini ad r prima di quelli più lontani, tale lista è una coda.

La BFS e descritta utilizzando gli operatori delle code:

```
procedure BFS(var G: grafo; u: nodo);
    var v: nodo, Q: coda;
    begin
        CREACODA(Q); INCODA(u, Q);
        while not CODAVUOTA(Q) do begin
            u := LEGGICODA(Q); FUORICODA(Q);
            {esamina il nodo u e marcalo "visitato"};
            for each v ∈ A(u) do begin
                {esamina l'arco (u, v)};
                if v non è marcato "visitato" and v ∉ Q then INCODA(v, Q);
            end;
        end;
    end;
```

La verifica di non appartenenza del nodo v alla coda può essere facilmente effettuata in tempo O(1) introducendo nella procedura BFS un opportuno vettore booleano in cui il bit v è impostato a vero quando v è inserito in Q ed è rimesso a falso quando v è estratto da Q. Questa verifica è necessaria per evitare che la coda contenga più di una copia di v, qualora v sia stato precedentemente inserito in coda durante l'esame di un arco (w, v) uscente da un nodo w già marcato visitato. Anche la "marcatura" di ciascun nodo può essere effettuata con un secondo vettore booleano, con ciascun bit u inizializzato a "falso", che poi viene impostato a "vero" quando viene visitato il nodo u.

Poiché ogni nodo del grafo è marcato una sola volta e, per ogni nodo v, viene scandito l'intero insieme $A(v)$ una sola volta, la complessità di entrambe le procedure è $\Theta(n + m)$.

Spiegare il concetto di collisione e le corrispondenti tecniche di gestione per dizionari

La realizzazione del dizionario con tabelle hash si basa sul concetto di ricavare direttamente dal valore della chiave la posizione che la chiave stessa dovrebbe occupare in un vettore. Con questa realizzazione, ciascuna operazione, pur avendo complessità $O(n)$ nel caso pessimo, richiede tempo $O(1)$ nel caso medio. Sia K l'insieme di tutte le possibili chiavi distinte e si memorizzi il dizionario in un vettore V di dimensione m. La soluzione ideale sarebbe quella di possedere una funzione d'accesso

$$H : K \to \{1, \ldots, m\}$$

che permetta di ricavare la posizione $H(k)$ della chiave k nel vettore V in modo che per ogni $k_1 \in K$ e $k_2 \in K$, con $k_1 \neq k_2$, risulti $H(k_1) \neq H(k_2)$. Per garantire tale biunivocità, ovviamente, m non può essere più piccolo di $|K|$.

Per esempio, utilizzando un vettore con $m = |K|$ posizioni, si potrebbe accedere direttamente alla posizione contenente la chiave in tempo $O(1)$. Se $|K|$ è grande, purtroppo, questo metodo è improponibile perché richiede uno spreco enorme di memoria.

Per evitare inutili sprechi di memoria, la dimensione m del vettore va scelta in base al numero di chiavi attese, cioè che ci si aspetta possano essere effettivamente memorizzate contemporaneamente, e non in base a quello di tutte le chiavi possibili, la maggior parte delle quali potrebbe non apparire mai nel dizionario. Occorre quindi rinunciare alla biunivocità della funzione H ammettendo che chiavi diverse possano avere lo stesso indirizzo nel vettore, cioè che $H(k_1) = H(k_2)$ anche se $k_1 \neq k_2$.

In tal caso, la soluzione estrema che minimizza l'uso della memoria sembra quella in cui c'è un unico indirizzo $H(k)$ per ogni possibile chiave k, per esempio mantenendo tutte le chiavi in un'unica lista. In questo modo, la memoria richiesta è linearmente proporzionale al numero n di chiavi presenti nel dizionario, ma anche il tempo per le operazioni cresce con un inaccettabile $O(n)$. Occorre pertanto una soluzione di compromesso, con m maggiore di 1 ma molto minore di $|K|$, che mantenga bassi sia il tempo di esecuzione delle operazioni (possibilmente $O(1)$, come nel caso $m = |K|$), sia l'occupazione di memoria (possibilmente $O(n)$ come nel caso $m = 1$).

Rinunciando alla biunivocità di H, sorge il nuovo problema di scegliere le posizioni per memorizzare due chiavi k_1 e k_2 che collidono, cioè con $H(k_1) = H(k_2)$, qualora siano entrambe presenti contemporaneamente nel vettore.

Quattro buoni metodi di generazione di indirizzi hash sono i seguenti. I primi tre presuppongono $m = 2^p$, per qualche p, mentre il quarto presuppone m dispari (meglio se primo):

a. $H(k) = int(b)$, dove b è un sottoinsieme di p bit di $bin(k)$, solitamente estratti nelle posizioni centrali;
b. $H(k) = int(b)$, dove b è dato dalla somma modulo 2, effettuata bit a bit, di diversi sottoinsiemi di p bit di $bin(k)$;
c. $H(k) = int(b)$, dove b è un sottoinsieme di p bit estratti dalle posizioni centrali di $bin(int(bin(k))^2)$;
d. $H(k)$ è uguale al resto della divisione di $int(bin(k))$ per m.

La funzione hash definita in (d) presuppone che m sia dispari, perché se m fosse uguale a 2^p allora due numeri con gli stessi p bit finali darebbero sempre luogo ad una collisione. Questa funzione è la migliore dal punto di vista probabilistico e fornisce un'eccellente distribuzione degli indirizzi $H(k)$ nell'intervallo $[0, m-1]$, specie se m è un numero primo non troppo vicino ad una potenza di 2 (o a qualsiasi altra base con la quale il calcolatore utilizzato rappresenti i numeri). Se il calcolatore rappresenta numeri usando la base 2, allora alcuni valori adeguati per m sono: 13, 23, 47, 97, 193, 383, 769, 1531, 3067, 6143, 12289, 24571.

Una soluzione vantaggiosa è l'uso di una tabella con liste di trabocco esempio:

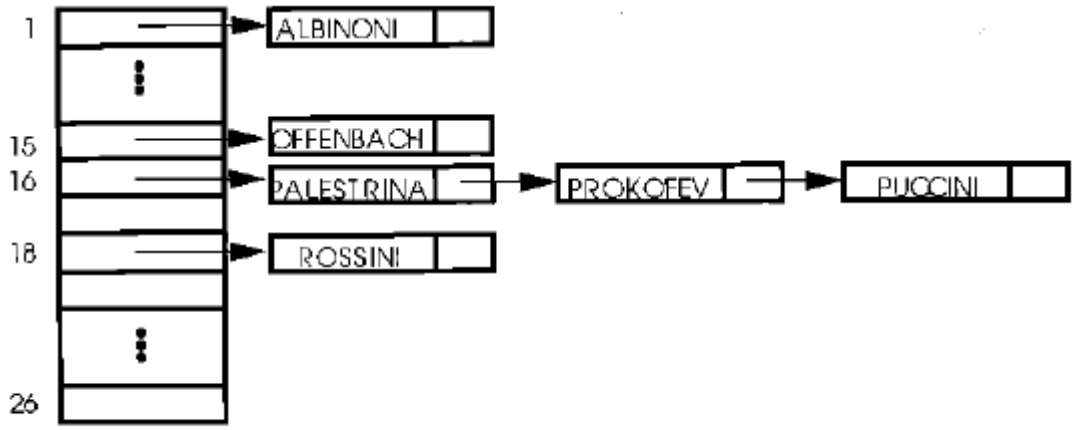


Figura 6.2: Liste di trabocco.

IN BREVE DIZIONARIO:
Una collisione hash è una situazione che avviene quando due diversi input producono lo stesso output tramite una funzione hash. Con la funzione hash, si vuole ricavare l'output (un numero intero) che sia scorrelato dalla struttura dell'input (la chiave stessa). Da qui il termine "hash", che in inglese significa "tritare" (e quindi "polpetta"): si sminuzza la chiave in pezzettini che sono rimescolati in modo da formare un intero che sia il più "casuale" Possibile. Ovviamente l'algoritmo che calcola la funzione hash non e casuale.
Il dizionario può essere gestito con un array, in modo sequenziale, ma se la dimensione è molto grande è uno spreco di memoria. Bisognerebbe già sapere il numero di chiave attese nel dizionario. Se due chiavi k1 e k2 producono lo stesso output, h(k1) ==h(k2) con k1 diverso da k2, si ha una collisione. L'Hash Table comunemente più implementata per ridurre il numero di collisioni, è con lista di Trabocco. Le liste di trabocco contengono tutte chiavi che collidono in un'unica lista. (Eliminazione, cancellazione, aggiornamento) è sufficiente calcolare l'hash della chiave da ricercare, accedere alla lista di trabocco e scandirla fino a trovare la chiave desiderata.

Fornire la specifica di problema di ricerca

***DEFINIZIONE: UN* PROBLEMA DI RICERCA P È UN PROBLEMA SPECIFICATO CON UNA QUINTUPLA DEL TIPO:**
<I,S,R,S ∪ {⊥},qric>
DOVE qric È LA REGOLA CHE DEFINISCE, IN BASE AD R, LA RELAZIONE Rqric CONTENENTE TUTTI E SOLI I SEGUENTI ELEMENTI: • *OGNI COPPIA CONTENUTA IN R;* • *UNA COPPIA* <i,⊥> *PER OGNI ISTANZA i DI P PER LA QUALE P NON HA SOLUZIONI.*

Spiegare la tecnica algoritmica greedy

Nella vita di tutti i giorni, spesso si adottano strategie che danno immediatamente buoni risultati, ma che a lungo andare si rivelano scadenti. È questa la strategia dell'ingordo (o "greedy"):

Si supponga di avere monete da 11, 5 e 1 centesimo, e di dover fare un resto di 15 centesimi. L'ingordo giudicherà la moneta da 11 più "appetibile" delle altre, e farà un resto con una moneta da 11 e quattro da 1, utilizzando cinque monete in tutto, mentre la soluzione ottima è data da tre monete da 5 centesimi!

Il metodo greedy si può applicare a quei problemi di ottimizzazione in cui occorre selezionare un sottoinsieme S "ottimo" di oggetti, che verificano certe proprietà, da un insieme dato $\{a_1, \ldots, a_n\}$ di n oggetti. Uno "scheletro" di procedura che adotta tale metodo è il seguente:

```
procedure GREEDY(insieme {a_1, ..., a_n} di oggetti);
    begin
        S := ∅;
        {ordina gli a_i per "appetibilità" decrescente};
        for i := 1 to n do
            if {a_i può essere aggiunto ad S}
                then S := S ∪ {a_i};
        {restituisci S come risultato}
    end;
```

Un algoritmo greedy ordina dapprima gli oggetti in base ad un criterio di "appetibilità". La soluzione del problema è poi costruita in modo incrementale considerando gli oggetti uno alla volta e aggiungendo ogni volta l'oggetto più appetibile, se possibile. In altri termini, l'algoritmo effettua una sequenza di scelte, preferendo ogni volta la scelta che fornisce immediatamente il miglior risultato. Fatta la scelta, è successivamente risolto un sottoproblema dello stesso tipo ma di dimensione più piccola.

Fornire la specifica sintattica e semantica degli operatori cancnodo e cancarco per la struttura dati grafo

SPECIFICA SINTATTICA CANCNODO CANCARCO:

CANCNODO: (nodo, grafo) → grafo
CANCARCO: (nodo, nodo, grafo) → grafo

SPECIFICA SEMANTICA CANCNODO CANCARCO:

CANCNODO$(u, G) = G'$

Pre: $G = (N, A)$, $u \in N$, non esiste alcun nodo v tale che $(u, v) \in A$ o $(v, u) \in A$

Post: $G' = (N', A)$, $N' = N - \{u\}$

Fornire la specifica sintattica e semantica degli operatori insSottoAlbero e insPrimoSottoAlbero per la struttura dati Alberi n-ari

SPECIFICA SINTATTICA INSOTTOALBERO, INSPRIMOSOTTOALBERO:

INSPRIMOSOTTOALBERO:(*nodo,albero,albero*)→*albero*
INSSOTTOALBERO: (*nodo, albero, albero*) → *albero*

SPECIFICA SEMANTICA INSOTTOALBERO INSPRIMOSOTTOALBERO:

INSPRIMOSOTTOALBERO(u, T, T') =T"
PRE: T≠Λ, T '≠Λ, u∈ N
POST: T" È OTTENUTO DA T AGGIUNGENDO L'ALBERO T' LA CUI RADICE r' È IL NUOVO PRIMOFIGLIO DI u

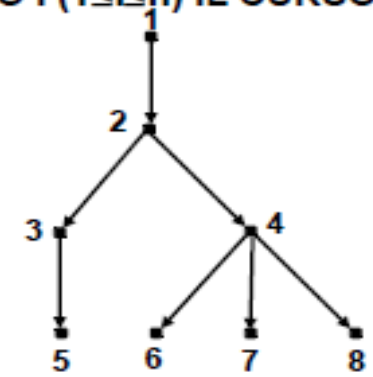
INSSOTTOALBERO(u, T, T') =T"
PRE: T≠Λ, T '≠Λ, u∈ N, u NON È RADICE DI T
POST: T" È L'ALBERO OTTENUTO DA T AGGIUNGENDO IL SOTTOALBERO T' DI RADICE r' (CIOÈ r' DIVENTA IL NUOVO FRATELLO CHE SEGUE u NELLA RELAZIONE D'ORDINE)

Spiegare la realizzazione di alberi n-ari mediante vettore dei padri, liste di figli e con cursori, fornendo Vantaggi e svantaggi di ognuna

ALBERO N-ARIO TRAMITE VETTORE DI PADRI CON VANTAGGI E SVANTAGGI:

RAPPRESENTAZIONE CON VETTORE DI PADRI

IMMAGINANDO DI NUMERARE I NODI DI T DA 1 A n, LA PIÙ SEMPLICE REALIZZAZIONE (SEQUENZIALE) CONSISTE NELL'USARE UN VETTORE CHE CONTIENE, PER OGNI NODO i (1≤i≤n) IL CURSORE AL PADRE.



1	0
2	1
3	2
4	2
5	3
6	4
7	4
8	4

È FACILE, COSÌ, VISITARE I NODI LUNGO PERCORSI CHE VANNO DA FOGLIE A RADICE. È, INVECE, PIÙ COMPLESSO INSERIRE E CANCELLARE SOTTOALBERI.

ALBERO N-ARIO TRAMITE LISTE DI FIGLI VANTAGGI E SVANTAGGI:

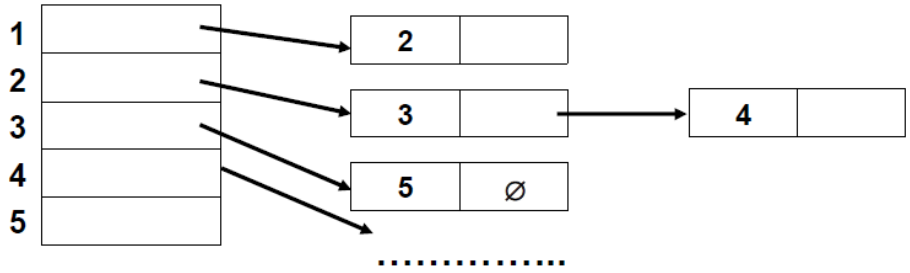
UNA VARIANTE MOLTO USATA, POICHÉ NON SI PUÒ ASSOCIARE AD OGNI ELEMENTO UN NUMERO DI PUNTATORI UGUALE AL MASSIMO DEI FIGLI, E'

LA RAPPRESENTAZIONE ATTRAVERSO LISTE DI FIGLI

COMPRENDE:

- **IL VETTORE DEI NODI, IN CUI, OLTRE ALLE EVENTUALI ETICHETTE DEI NODI, SI MEMORIZZA IL RIFERIMENTO INIZIALE DI UNA LISTA ASSOCIATA AD OGNI NODO;**
- **UNA LISTA PER OGNI NODO, DETTA *LISTA DEI FIGLI*. LA LISTA ASSOCIATA AL GENERICO NODO i CONTIENE TANTI ELEMENTI QUANTI SONO I SUCCESSORI DI i; CIASCUN ELEMENTO È IL RIFERIMENTO AD UNO DEI SUCCESSORI.**

PER L'ESEMPIO PRECEDENTE SI AVREBBE:



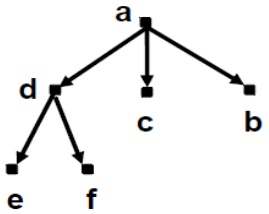
Ovviamente, la realizzazione con liste dei figli presenta il vantaggio di poter scorrere velocemente tutti i figli di un dato nodo, ma non permette di realizzare efficientemente altri operatori come PADRE o SUCCFRATELLO. Infatti, per poter individuare il padre o il fratello successivo di un dato nodo u, occorre scandire nel caso pessimo tutte le n liste alla ricerca dell'unica lista nella quale compare il nodo u.

RAPPRESENTAZIONE MEDIANTE LISTA PRIMOFIGLIO/FRATELLO

PREVEDE LA GESTIONE DI UNA LISTA E QUESTO PUÒ ESSERE FATTO IMPONENDO CHE TUTTI GLI ALBERI *CONDIVIDANO UN'AREA COMUNE* (AD ESEMPIO UN VETTORE NELLA REALIZZAZIONE CON CURSORI) E CHE *OGNI CELLA CONTENGA ESATTAMENTE DUE CURSORI: UNO AL PRIMOFIGLIO ED UNO AL FRATELLO SUCCESSIVO*. LA REALIZZAZIONE È SIMILE A QUELLA PROPOSTA PER GLI ALBERI BINARI CON L'UNICA DIFFERENZA CHE IL CURSORE NEL TERZO CAMPO PUNTA AL FRATELLO. NATURALMENTE E' POSSIBILE ANCHE PREVEDERE UN CURSORE AL *GENITORE*:

INIZIO: 4

	FIGLIO	NODO	FRATELLO
1	0	e	2
2	0	f	0
3	0	c	5
4	7	a	0
5	0	b	0
6			
7	1	d	3
8			



Fornire in pseudocodice l'algoritmo di ricerca in Profondità (DFS) per alberi n-ari

Nella teoria dei grafi, Ricerca in profondità, in inglese depth-first search (DFS), è un algoritmo di ricerca su alberi e grafi. A differenza della ricerca in ampiezza, ha la caratteristica di essere intrinsecamente ricorsivo.

Nel seguito, la DFS è descritta con una procedura Pascal ricorsiva, perchè più concisa ed elegante di una iterativa che utilizzi una pila, mentre la BFS è descritta utilizzando gli operatori delle code [cfr. § 3]. Le chiamate delle procedure sono: DFS(G, r) e BFS(G, r).

```
procedure DFS(var G: grafo; u: nodo);
    var v: nodo;
    begin
        {esamina il nodo u e marcalo "visitato"};
        for each v ∈ A(u) do begin
            {esamina l'arco (u, v)};
            if v non è marcato "visitato" then DFS(G, v);
        end;
    end;
```

Descrivere come uno stack e una coda possano essere rappresentati mediante vettore

STACK(PILA) VETTORE:

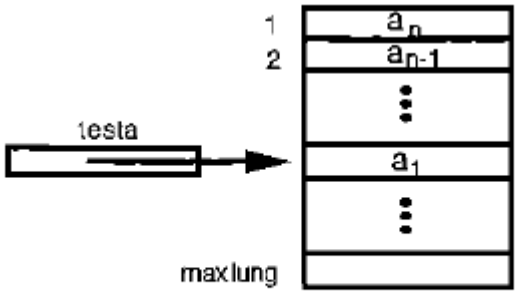


Figura 3.3: Realizzazione di una pila con un vettore.

Come mostrato nella Fig. 3.3, una realizzazione con vettore del tipo di dato pila consiste nel memorizzare gli n elementi della pila, in ordine inverso, nelle prime n posizioni di un vettore di *maxlung* elementi, mantenendo un cursore alla testa della pila. La pila vuota viene individuata dal valore 0 contenuto nel cursore, mentre la pila satura, cioè che contiene il massimo numero consentito di elementi, è individuata dal valore *maxlung*. FUORIPILA è realizzata decrementando il valore del cursore, mentre INPILA è realizzata incrementando il valore del cursore ed inserendo il nuovo elemento nella posizione del vettore individuata dal cursore. ~~Eventuali tentativi di~~

Con questa realizzazione, ogni operazione della pila richiede tempo costante per essere eseguita. Per ottenere la stessa complessità anche per tutte le operazioni della coda, si utilizza un vettore "circolare".

CODA COME VETTORE CIRCOLARE:

3.3 Realizzazione di una coda con vettore circolare

Si supponga di avere un array di *maxlung* elementi, indicati da 0 a *maxlung* − 1, in cui l'elemento di indice 0 è considerato come successore di quello di indice

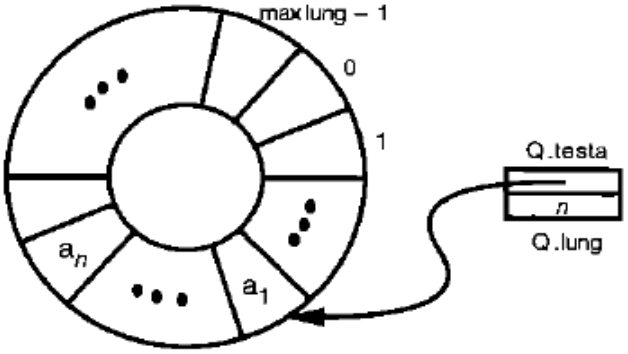


Figura 3.4: Realizzazione di una coda con un vettore circolare.

maxlung − 1. La coda è contenuta in n posizioni consecutive del vettore, per esempio secondo il verso orario, e i suoi elementi estremi sono individuati da due cursori. Per cancellare o per inserire un elemento, viene incrementato di uno (modulo *maxlung*) il rispettivo cursore di testa o di fondo. In questo modo la coda "migra" lungo il vettore circolare e, ad ogni istante, si trova da qualche parte nel vettore, sempre in posizioni consecutive secondo il verso orario.

Anziché utilizzare due cursori, per individuare la coda è sufficiente conoscere l'indice del primo elemento e la lunghezza della coda stessa, come mostrato nella Fig. 3.4. Infatti, se i è la posizione dell'elemento in testa alla coda, allora quella dell'ultimo elemento della coda è $(i + n - 1)$ modulo *maxlung*.

positivo o negativo, del confronto tra la soluzione parziale e i vincoli posti dal problema alla natura delle soluzioni.

Questa tecnica, detta backtrack, è alla base di molti algoritmi di visita di alberi e grafi, ed è usata per generare sistematicamente tutte le soluzioni ammissibili di un problema ("to backtrack" significa "tornare sui propri passi").

Certo, una tecnica non molto astuta, ma può rivelarsi molto utile.

Motivare le differenze fra una tecnica greedy e una di backtracking quando applicate ad un problema
Di ottimizzazione

Problemi di ottimizzazione nei quali alle soluzioni ammissibili è associata una misura (o costo, o obiettivo) e si vuole trovare una soluzione ottima, cioè una soluzione ammissibile la cui misura sia minima o massima (ad esempio: trovare una foglia di un albero radicato il cui livello sia massimo).

Quando si applica il backtracking o la tecnica greedy in un problema di ottimizzazione bisogna stabilire il tipo di output aspettato, sempre in base al quesito.

Per esempio, l'algoritmo greedy è utile per ottenere una delle possibili soluzioni nel più tempo minore possibile. Anche se questa è la tecnica più veloce non trova la soluzione ottimale nell'insieme delle soluzioni.

Invece l'algoritmo di backtracking è decisamente più lento rispetto a quello greedy, tuttavia seleziona la soluzione migliore dell'insieme delle soluzioni.

Se in un problema di ottimizzazione, si pone come obiettivo il tempo di ricerca, la migliore è la tecnica greedy. Se invece si vuole ottenere tra tutte le soluzioni, quella più vantaggiosa allora bisogna prediligere il backtracking a discapito del tempo di ricerca della soluzione.

Fornire la specifica sintattica e semantica degli operatori costrBinAlbero e cancSottoBinAlbero per la struttura dati Alberi binari

SINTATTICA

COSTRBINALBERO : (ALBEROBIN,ALBEROBIN) → ALBEROBIN

CANCSOTTOBINALBERO : (NODO,ALBEROBIN) → ALBEROBIN

SEMANTICA

COSTRBINALBERO(T,T') = T''

POST: T'' SI OTTIENE DA T E DA T' INTRODUCENDO AUTOMATICAMENTE UN NUOVO NODO r'' (RADICE DI T'') CHE AVRA' COME SOTTOALBERO SINISTRO T E SOTTOALBERO DESTRO T' (*SE T = Λ E T' = Λ, L'OPERATORE INSERISCE LA SOLA RADICE r''; SE T = Λ, r'' NON HA FIGLIO SINISTRO; SE T' = Λ, r'' NON HA FIGLIO DESTRO*)

CANCSOTTOBINALBERO(u,T) = T'

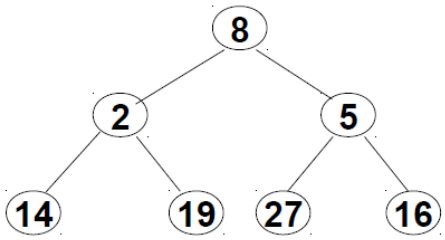
PRE: T ≠ Λ, u ∈ N

POST: T' E' OTTENUTO DA T ELIMINANDO IL SOTTOALBERO DI RADICE u, CON TUTTI I SUOI DISCENDENTI

VALIDA PER ALBERI DI OGNI ORDINE AGISCE POTANDO DAL NODO u.

Spiegare la realizzazione di alberi binari mediante vettore e collegata con cursori, fornendo vantaggi e svantaggi di ognuna

UNA POSSIBILE RAPPRESENTAZIONE DI UN ALBERO BINARIO E' QUELLA SEQUENZIALE MEDIANTE VETTORE. LA RADICE E' IN PRIMA POSIZIONE; PER IL GENERICO NODO p MEMORIZZATO IN POSIZIONE i, SE ESISTE IL FIGLIO SINISTRO E' MEMORIZZATO IN POSIZIONE 2*i, SE ESISTE IL FIGLIO DESTRO E' MEMORIZZATO IN POSIZIONE 2*i+1



1	8
2	2
3	5
4	14
5	19
6	27
7	16

Se un albero è incompleto rappresenta uno spreco di memoria e per avvalorare quel nodo vuoto dell'array bisogna aggiungere il campo booleano true se presente altrimenti false.

1	VERO	8
2	FALSO	24
3	VERO	5
4	FALSO	62
5	FALSO	3
6	VERO	27
7	VERO	16

TUTTAVIA E' IMMEDIATO VERIFICARE CHE:

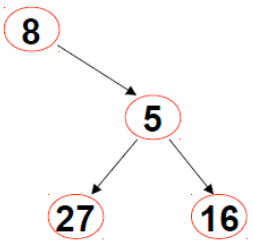
- **ALBERI BINARI NON COMPLETI VENGONO RAPPRESENTATI CON SPRECO DI MEMORIA**
- **E' IMPOSTO UN LIMITE MASSIMO PER IL NUMERO DI NODI DELL'ALBERO**
- **LE OPERAZIONI DI AGGIUNTA ED ELIMINAZIONE DI NODI O DI SOTTOALBERI COMPORTANO DIVERSI SPOSTAMENTI NELL'ARRAY**

LA RAPPRESENTAZIONE COLLEGATA DI UN ALBERO

3

INIZIO

	SIN	NODO	DES
1	0	16	0
2			
3	0	8	7
4			
5	0	27	0
6			
7	5	5	1



UN PRIMO METODO RICHIEDE DI UTILIZZARE UN ARRAY, IN MODO CHE AD OGNI NODO DELL'ALBERO CORRISPONDA UNA COMPONENTE DELL'ARRAY IN CUI SONO MEMORIZZATE LE INFORMAZIONI (NODO, RIFERIMENTO AL FIGLIO SINISTRO, RIFERIMENTO AL FIGLIO DESTRO).

IL RIFERIMENTO E' IL VALORE DELL'INDICE IN CORRISPONDENZA DEL QUALE SI TROVA LA COMPONENTE CHE CORRISPONDE AL FIGLIO SINISTRO O DESTRO.

SE IL FIGLIO NON ESISTE IL RIFERIMENTO HA VALORE 0.

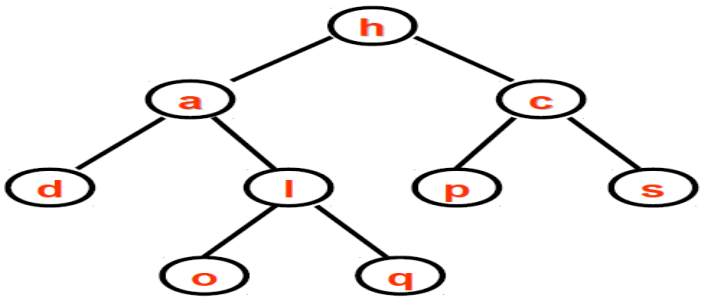
Fornire in pseudocodice l'algoritmo di visita in post-ordine per alberi binari

VISITA POSTORDINE L'ALBERO BINARIO T

SE L'ALBERO NON E' VUOTO
ALLORA
1-VISITA IN POSTORDINE IL SOTTOALBERO SINISTRO DI T
2-VISITA IN POSTORDINE IL SOTTOALBERO DESTRO DI T
3-ANALIZZA LA RADICE
FINE

ESEMPIO:

SIA UN ALBERO BINARIO CHE HA DEI CARATTERI NEI NODI



LA VISITA IN PREORDINE: h a d l o q c p s

LA VISITA IN POSTORDINE: d o q l a p s c h

LA VISITA SIMMETRICA: d a o l q h p c s

Descrivere come un albero binario può essere utilizzato per rappresentare una coda con priorità, descrivendo inoltre come si effettuano le operazioni di inserimento e cancellazione di elementi

RAPPRESENTAZIONE CON ALBERI BINARI

GLI ELEMENTI DI UNA CODA CON PRIORITÀ C POSSONO ESSERE MEMORIZZATI NEI NODI DI UN ALBERO BINARIO B CHE DEVE AVERE LE PROPRIETÀ:

- **L'ALBERO B È *QUASI PERFETTAMENTE BILANCIATO***
  - **SE k È IL LIVELLO MASSIMO DELLE FOGLIE, ALLORA B HA ESATTAMENTE 2^K-1 NODI DI LIVELLO MINORE DI k ;**
  - **TUTTE LE SUE FOGLIE DI LIVELLO k SONO ADDOSSATE A SINISTRA;**
- **L'ALBERO B È *PARZIALMENTE ORDINATO***
  - **OGNI NODO CONTIENE UN ELEMENTO DI C CHE E' MAGGIORE DI QUELLO DEL PADRE**

PER REALIZZARE GLI OPERATORI SI OSSERVI CHE:

- **MIN RESTITUISCE IL CONTENUTO DELLA RADICE DELL'ALBERO B;**
- **INSERISCI DEVE INSERIRE UNA NUOVA FOGLIA IN MODO DA MANTENERE VERIFICATA LA PROPRIETÀ (1) E QUINDI FAR "SALIRE" L'ELEMENTO INTRODOTTO FINO A VERIFICARE LA PROPRIETÀ (2);**
- **CANCELLAMIN PREVEDE LA CANCELLAZIONE DELLA FOGLIA DI LIVELLO MASSIMO PIÙ A DESTRA, IN MODO DA MANTENERE VERIFICATA LA PROPRIETÀ (1) ED IL REINSERIMENTO DEL CONTENUTO DELLA FOGLIA CANCELLATA NELL'ALBERO PARTENDO DALLA RADICE E FACENDOLO "SCENDERE" IN MODO CHE L'ALBERO COSÌ MODIFICATO VERIFICHI ANCHE LA PROPRIETÀ (2).**

Fornire la specifica di problema di ottimizzazione

PROBLEMI DI OTTIMIZZAZIONE/1

ALLE SOLUZIONI AMMISSIBILI È' ASSOCIATA UNA MISURA (COSTO, OBIETTIVO): RISOLVERE IL PROBLEMA NON SIGNIFICA TROVARE UNA QUALUNQUE SOLUZIONE, MA LA MIGLIORE SOLUZIONE SECONDO LA MISURA O CRITERIO DI PREFERENZA FISSATO.

DEFINIZIONE

UN PROBLEMA DI OTTIMIZZAZIONE P È' UN PROBLEMA SPECIFICATO CON UNA QUINTUPLA DEL TIPO:

$$<I,S,R,S \cup \{\perp\},q_{ott}(M,m,\subseteq)>$$

- **M È' INSIEME QUALSIASI;**
- **m È' UNA FUNZIONE DEL TIPO $I \times S \rightarrow M$ DETTA *FUNZIONE OBIETTIVO* DI P; PER UNA CERTA ISTANZA $i \in I$, IL VALORE $m(i,s)$ RAPPRESENTA UNA MISURA DELL'ELEMENTO S NELLO SPAZIO DELLE SOLUZIONI**
- **$\subseteq$ È UNA RELAZIONE DI ORDINAMENTO SU M ($x \subseteq y$ "x MIGLIORE DI y");**
- **$q_{ott}(M,m,\subseteq)$ È UNA REGOLA CHE DEFINISCE, IN BASE AD R, LA RELAZIONE $R_{q_{ott}} \subseteq I \times S$ SU I ED S COSÌ DA INDIVIDUARE:**
  - ***a)*** **UNA COPPIA (i,s) $\forall i \ni' \neg\exists$ s' ASSOCIATA AD i MIGLIORE;**
  - ***b)*** **UNA COPPIA $(i,\perp)$ $\forall i$ PER LA QUALE P NON HA SOLUZIONI.**

Facendo riferimento ad uno specifico problema di ricerca spiegare ed illustrare l'esecuzione di una Strategia di backtracking applicata ad una istanza di quel problema.

ESEMPIO:

PROBLEMA DELLE 8 REGINE CON ALGORITMO DI BACKTRACKING.

COME SI È DETTO POSSIAMO ESPRIMERE LA SOLUZIONE ATTRAVERSO UNA PERMUTAZIONE DEI NUMERI DA 1 A 8. LA SOLUZIONE SI PUÒ ESPRIMERE MEDIANTE IL VETTORE V DEFINITO COME SEGUE:

```
TYPE    POSIZIONE=[1..8];
        VETTORE = ARRAY [1..8] OF POSIZIONE;
VAR  V:VETTORE;
OTTO_REGINE (V:VETTORE per riferimento; SUCCESSO:BOOLEAN)

K ← 1
V[k] ← 1
SUCCESSO ← FALSE
repeat
   VERIFICA(a); /*PROGRAMMA CHE CONTROLLA IL VETTORE*/
   if (a) then  /*SI CERCA UNA NUOVA POSIZIONE/*
     repeat
       if (V[k]<8) then BEGIN
         b ← TRUE, V[k] ← V[k] +1
       else /*BACKTRACK/*
         b ← FALSE, k ← k-1
     until b or (k=0)
   else if (k=8) then
      SUCCESSO ← TRUE
   else  /*SI AGGIUNGE COMPONENTE/*
     k ← k+1, V[k] ←  1
until SUCCESSO or (k=0)
```

I VINCOLI IMPOSTI SONO :

i. **LE COMPONENTI DI V DEVONO COSTITUIRE UNA PERMUTAZIONE DA 1 A 8; PER OGNI COPPIA DI INDICI i E j VALE V[i] ≠ V[j]**

ii. **NON È POSSIBILE AVERE DUE ELEMENTI SULLA STESSA DIAGONALE, CIOE'**

$$(i - j) \neq (k - 1)$$
$$(i + j) \neq (k + 1)$$

IL SOTTOPROGRAMMA *VERIFICA* HA IL COMPITO DI ESAMINARE IL VETTORE V E SE IL VETTORE VIOLA UNO DEI VINCOLI RENDE *TRUE* LA VARIABILE a, ALTRIMENTI SE I VINCOLI SONO AMBEDUE SODDISFATTI a = *FALSE*.

Spiegare cosa è un problema di ottimizzazione e quali tecniche si possono adottare per risolverlo

Problemi di ottimizzazione nei quali alle soluzioni ammissibili è associata una misura (o costo, o obiettivo) e si vuole trovare una soluzione ottima, cioè una soluzione ammissibile la cui misura sia minima o massima (ad esempio: trovare una foglia di un albero radicato il cui livello sia massimo).

Tecnica di progetto. Successivamente, si cerca di applicare certe tecniche di progetto di algoritmi per rendere gli algoritmi più veloci. Le cinque tecniche principali sono: divide et impera, backtrack, greedy, programmazione dinamica e ricerca locale. La prima di queste tecniche è già stata implicitamente introdotta, e consiste nel partizionare il problema in sottoproblemi più piccoli, risolubili indipendentemente, le cui soluzioni possono essere ricombinate per ottenere la soluzione del problema di partenza (ad esempio: la procedura di "ricerca binaria"

Riportare l'albero binario di ricerca corrispondente ad una coda con priorità dove aver inserito nell'ordine
I seguenti elementi: 17, 26, 8, 50, 16, 19, 93, 4, 9 e 55. Poi illustrare il processo di inserimento del 7.
Dopo l'inserimento del 7 illustrare il processo di rimozione del minimo.

0) Albero, 4

Inserir. 7 8 9

16 17 19 26

50 55 93 “7”

Il 7 viene inserito come figlio dx di 17. Rispettando la coda con priorità e quindi la sua relazione d’ordine si fa salire il 7 fino a quando…

1) Scalata 4

8 9

16 “7” 19 26

50 55 93 17

2) 4

“7 “ 9

16 8 19 26

50 55 93 17

3) Cancellazione min.:

-

“7 “ 9

16 8 19 26

50 55 93 17

4) Il minimo dei due figli diventa min. dell’albero:

“7”

8 9

16 17 19 26

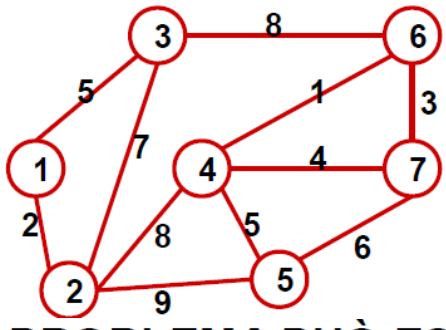
50 55 93

Facendo riferimento ad uno specifico problema di ottimizzazione spiegare ed illustrare l'esecuzione di Una strategia greedy applicata ad una istanza di quel problema.

UN ALTRO PROBLEMA:

PROBLEMA DEL MINIMO ALBERO DI COPERTURA

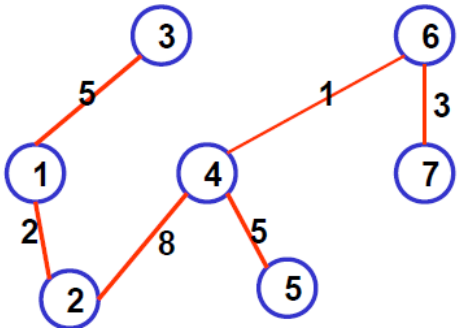
DATO UN GRAFO NON ORIENTATO E CONNESSO G=(N,A), CON PESI SUGLI ARCHI (NON NEGATIVI), TROVARE UN ALBERO DI COPERTURA PER G, CIOÈ UN ALBERO AVENTE TUTTI I NODI IN N, MA SOLO ALCUNI ARCHI IN A, IN MODO TALE CHE SIA MINIMA LA SOMMA DEI PESI ASSOCIATI AGLI ARCHI.



IL PROBLEMA PUÒ ESSERE RISOLTO CON MOLTI ALGORITMI, DEI QUALI I PIU' NOTI SI DEVONO A KRUSKAL (1956) E A PRIM (1957).

L'ALGORITMO DI KRUSKAL USA LA *TECNICA "GREEDY"*.

L'ALBERO DI COPERTURA MINIMO PER IL GRAFO PRECEDENTE E'



L'ALGORITMO DI KRUSKAL, DOPO AVER ORDINATO GLI ARCHI SECONDO I PESI CRESCENTI, LI ESAMINA IN TALE ORDINE, INSERENDOLI NELLA SOLUZIONE *SE NON FORMANO CICLI CON ALTRI ARCHI GIÀ SCELTI*.

AD UN LIVELLO MOLTO GENERALE, L'ALGORITMO E' ESPRIMIBILE IN QUESTI TERMINI:

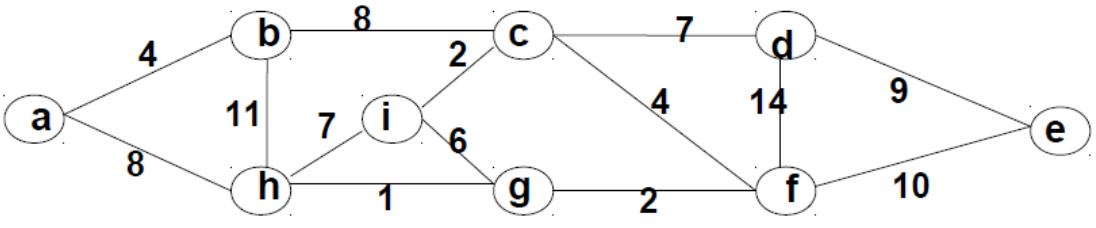
```
KRUSKAL(GRAFO)
  T ← Λ
  ORDINA GLI ARCHI DI G PER PESO CRESCENTE
  for a ← 1 to m do
    if (L'ARCO a=(i,j) NON FORMA CICLO
              CON ALTRI ARCHI DI T) then
      T ← T ∪ (a)
```

POSSIAMO RAPPRESENTARE IL GRAFO G COME PREFERIAMO. NELLA REALIZZAZIONE DATA DI SEGUITO IL GRAFO E' REALIZZATO CON UN VETTORE DI ARCHI E L'ALBERO T CON UNA LISTA DI ARCHI (REALIZZATA CON PUNTATORI).

Definizione dei tipi:

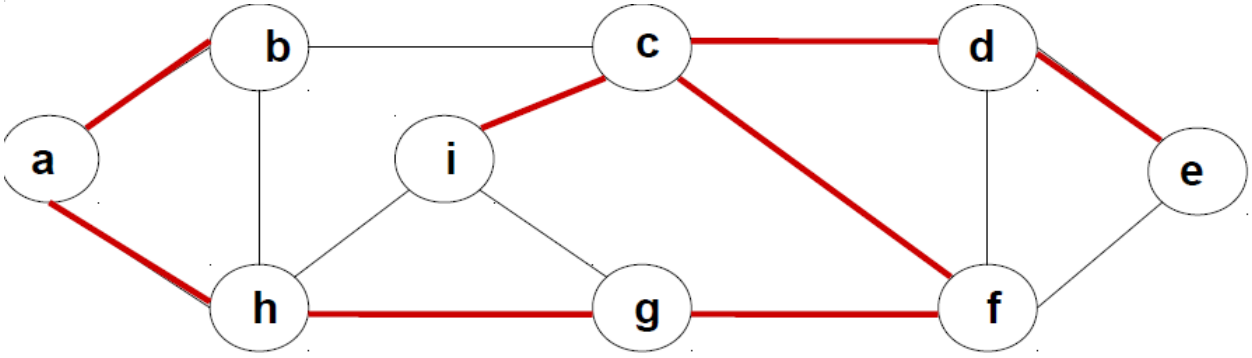
```
ARCO: tipo strutturato con componenti
  - i,j:INTEGER
  - PESO: INTEGER

GRAFO: tipo strutturato con componenti
  - A: ARRAY di MAXLUNG elementi di tipo ARCO
  - n,m:INTEGER
```



ECCO L'ORDINE IN CUI L'ALGORITMO CONSIDERA GLI ARCHI.

UNA VOLTA CHE SI CONSIDERA L'ARCO SE QUESTO UNISCE DUE ALBERI DISTINTI NELLA FORESTA, L'ARCO VIENE AGGIUNTO ALLA FORESTA E I DUE ALBERI FUSI IN UNO.



NELL'ALGORITMO DI KRUSKAL LA COSTRUZIONE DI T AVVIENE PER UNIONE DI COMPONENTI CONNESSE RAPPRESENTABILI COME INSIEMI DISGIUNTI (DUE COMPONENTI CONNESSE SI FONDONO IN UNA CON L'AGGIUNTA DI UN NUOVO ARCO)

COME E' NOTO, LA STRUTTURA MFSET (MERGE-FIND SET) È UNA PARTIZIONE DI UN INSIEME IN SOTTOINSIEMI DISGIUNTI DETTI COMPONENTI.

DUNQUE E' POSSIBILE UTILIZZARE UNA STRUTTURA DATI DI TIPO MFSET PER COSTRUIRE T.

```
KRUSKAL(G: GRAFO per riferimento)
Variabili:
  h: INTEGER;
  T: LISTA;
  S: MFSET; {PARTIZIONE DI UN INSIEME IN
                          SOTTOINSIEMI DISGIUNTI(COMPONENTI)}

                                                    CREALISTA(T)
  ORDINA G.A(1),…,G.A(G.m) PER ORDINE CRESCENTE DI G.A(h).PESO
  CREAMFSET(G.n,S)
  for h ← 1 to G.m do BEGIN
     if not TROVA (A(h).i,A(h).j,S) then
FONDI (A(h).i,A(h).j,S)
       INSLISTA (PRIMOLISTA(T),A(h),T)
```

Spiegare la strategia di risoluzione per il problema della ricerca del cammino minimo in un grafo
Adottata da un algoritmo a scelta del candidato

SI BASA SULL'IDEA DI CALCOLARE, IN ORDINE CRESCENTE, LA **LUNGHEZZA DEI CAMMINI MINIMI DA r A TUTTI I NODI DEL GRAFO**.

INDICHIAMO CON **S** L'INSIEME DEI NODI DI CUI, AD UN DATO ISTANTE, SI È GIÀ CALCOLATO LA LUNGHEZZA DEL CAMMINO MINIMO DA **r**.

UTILIZZIAMO UN VETTORE **DIST** CON TANTE COMPONENTI QUANTI SONO I NODI DEL GRAFO, IN MODO CHE **DIST(i)** RAPPRESENTI LA LUNGHEZZA DEL CAMMINO MINIMO TRA QUELLI CHE VANNO DA **r** A **i** PASSANDO SOLO PER NODI CONTENUTI IN **S** (A PARTE i STESSO). L'IPOTESI DI FONDO E' CHE LE **DISTANZE SIANO INTERI POSITIVI**.

OSSERVIAMO CHE SE IL PROSSIMO CAMMINO MINIMO DA GENERARE **C** È DA **r** AL NODO **u**, TUTTI I NODI SONO IN **S**. INFATTI SE UN NODO **k** DI **C** NON APPARTENESSE A **S** VI SAREBBE UN CAMMINO DA **r** A UN NODO k NON CONTENUTO IN **S** DI LUNGHEZZA MINORE A QUELLA DI **C**, CONTRADDICENDO L'IPOTESI CHE IL PROSSIMO CAMMINO DA GENERARE SIA **C**.

LA LUNGHEZZA DI **C** E IL NODO **u** SONO FACILMENTE INDIVIDUABILI; BASTA CALCOLARE IL VALORE MINIMO DI **DIST(i) PER i∉S**.

INDIVIDUATO **u** SI INSERISCE IN **S** E **SI AGGIORNA DIST** PER I NODI CHE **∉S** .

IN PARTICOLARE, SE PER UN CERTO NODO **z** CONNESSO A **u** DA **<u,z>** CON ETICHETTA **E**, LA SOMMA **DIST(u)+E** È MINORE DI **DIST(z)** ALLORA A **DIST(z)** VA ASSEGNATO IL NUOVO VALORE **DIST(u)+E**.

VIENE GENERATO UN *ALBERO DI COPERTURA T*, RADICATO IN r, CHE INCLUDE UN CAMMINO DA r AD OGNI ALTRO NODO.

L'ALBERO RADICATO T PUÒ ESSERE RAPPRESENTATO CON UN VETTORE DI PADRI, INIZIALIZZATO AD UN *ALBERO "FITTIZIO"* IN CUI TUTTI I NODI SONO FIGLI DI r CONNESSI AD UN *ARCO FITTIZIO ETICHETTATO* CON UN VALORE MAGGIORE DI TUTTE LE ALTRE ETICHETTE (MAXINT).

UNA SOLUZIONE AMMISSIBILE T È OTTIMA SE E SOLO SE

$$DIST(i) + C_{ij} = DIST(j) \ \forall \ (i,j) \in T \quad E$$

$$DIST(i) + C_{ij} \geq DIST(j) \ \forall \ ARCO(i,j) \in A$$

```
CAMMINIMINIMI (G: GRAFO per riferimento; r:NODO)

  CREAINSIEME(S)
  T(r) ← 0, DIST(r) ← 0
  for k ← 1 to n do
     if k≠r then
       T(k) ← r
       DIST(k) ← MAXINT
    INSERISCI(r,S)
    while not INSIEMEVUOTO(S) do
       i ← LEGGI(S)
       CANCELLA(i,S)
       for j∈A(i) do /*A(i) È L'INSIEME DI ADIACENZA*/
         if DIST(i) + Cij < DIST(j) then
           T(j) ← i
           DIST(j) ← DIST(i) + Cij
           if not APPARTIENE(j,S) then
             INSERISCI (j,S)
```

DIVIDE ET IMPERA ALGO NUM CONSECUTIVI

Data una sequenza di n numeri interi $(x_1, \ldots, x_n)$ diciamo che (x_i, x_{i+1}) è una coppia di numeri consecutivi se $x_{i+1} = x_i + 1$. Ad esempio nella sequenza $(12, 13, 24, 25, 26, 35, 67)$ ci sono 3 coppie di numeri consecutivi: $(12, 13)$, $(24, 25)$ e $(25, 26)$. Scrivere in pseudocodice un algoritmo che utilizzi la tecnica *divide-et-impera* e che calcoli quante coppie di numeri consecutivi sono contenute in una sequenza di n numeri interi $(x_1, \ldots, x_n)$ ricevuta in input. [7pt]

-NUM SUCC. ALGORITMO DIVIDE ET IMPERA (INPUT: ARRAY[])

BEGIN

INT DIM = DIM ARRAY

I=0

MENTRE I<DIM

SE ARRAY[i] == SUCCESSIVO IN ARRAY -1

INSERISCI ARRAY[I] E SUCCESSIVO IN ARRAY IN INSIEME

INCREMENTA i

FINE

RETURN INSIEME

- SUCCESSIVO IN ARRAY -1 (INPUT ELEM, ARRAY)

INT DIM = DIM ARRAY

Int I=0

MENTRE i<DIM

SE ELEMENTO IN POS I E' = ELEM+1

RESTITUISCI ELMEN IN POS i+1

ALTRIMENTI SE i==DIM-1

RESTITUISCI NIL(NIL=-1)