Tecniche avanzate di programmazione

Prof. Luca Campion

A.S. 2022/2023

Indice

Capitolo 1

Gli algoritmi greedy

1.1 Problemi di ottimizzazione

Gli algoritmi greedy sono procedure utilizzate per risolvere problemi di ottimizzazione.

Definizione 1.1. Un *problema di ottimizzazione* ha la forma:

$$\begin{aligned} \max / \min \quad & f(x) \\ & x \in K \end{aligned}$$

Questi problemi hanno spesso diverse soluzioni *ammissibili* (punti di K), ma poche di queste sono *ottime* (punti di K che massimizzano o minimizzano f).

L'idea che sta alla base degli algoritmi greedy è quella di *prendere la decisione localmente migliore*. Consideriamo il seguente problema:

Problema 1.2. Un alpinista si trova in una valle e vuole raggiungere la vetta più alta. A causa della nebbia riesce a vedere solo le vette adiacenti.

Un algoritmo greedy per risolvere questo problema potrebbe essere:

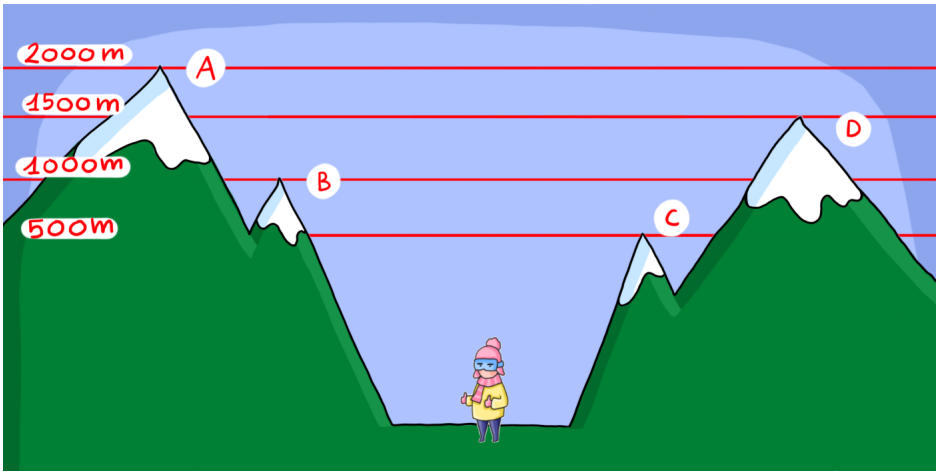
Algoritmo 1.1.1: Algoritmo greedy per il problema della vetta di altezza massima

```
1 while Ci sono vette vicine più alte del punto corrente do
2     sali sulla vetta più alta che riesci a vedere;
```

Questo è un problema di ottimizzazione: i punti ammissibili dell'insieme K sono le vette e le valli del profilo montuoso, mentre la funzione obiettivo f è quella che assenga alla vetta o valle l'altezza. Per alcune istanze del

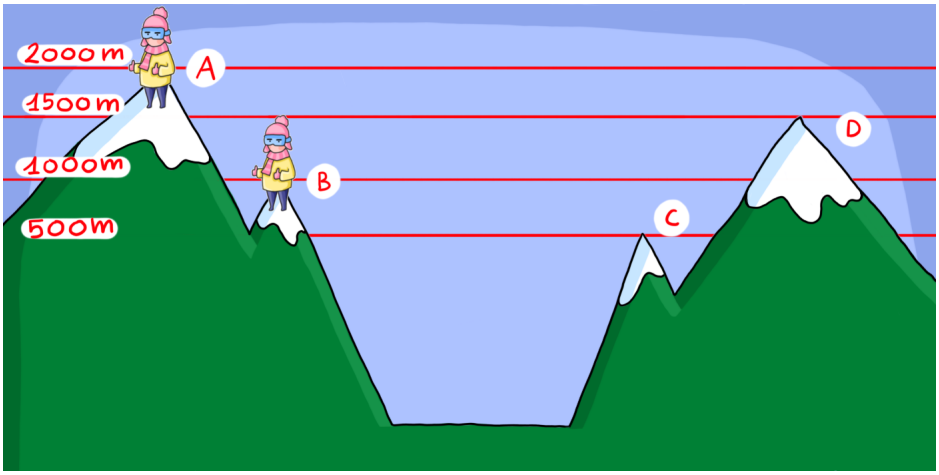
problema questo algoritmo porta alla soluzione ottima. Ad esempio, nella figura

Figura 1.1: Istanza del problema della vetta massima.



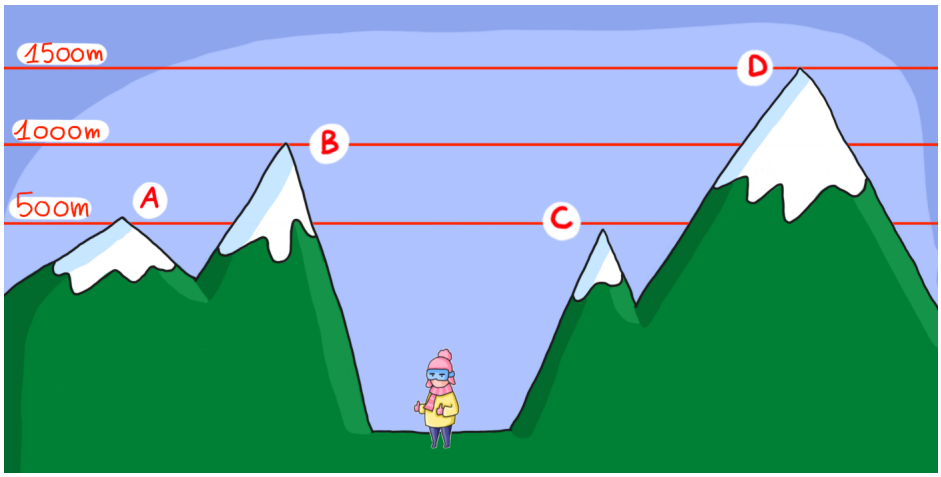
l'algoritmo 1 produce la soluzione ottima del problema, infatti l'alpinista passa per i punti B e A.

Figura 1.2: Istanza del problema della vetta massima.



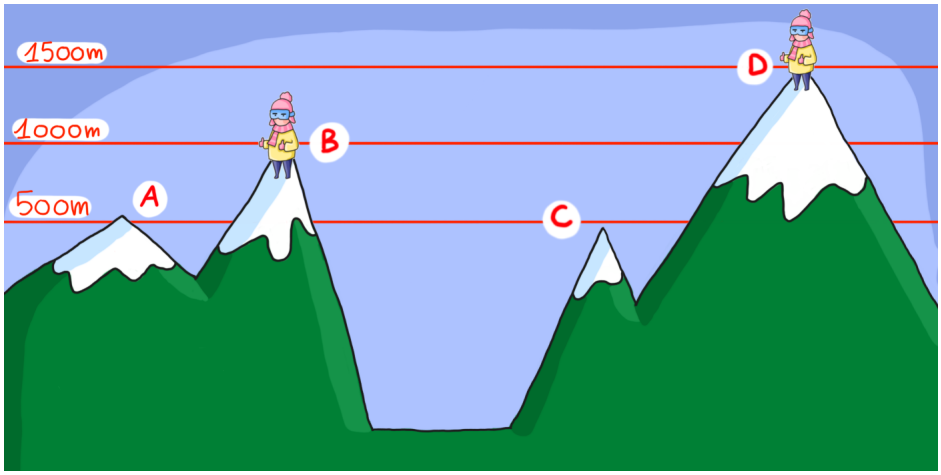
Se però consideriamo l'istanza in figura

Figura 1.3: Istanza del problema della vetta massima.



l'algoritmo 1 produce una soluzione ammissibile, ma non ottima, infatti l'alpinista passa al B e si ferma più in basso del punto D.

Figura 1.4: Istanza del problema della vetta massima.



Da questo esempio notiamo che:

Osservazione 1.3. Gli algoritmi greedy non sempre producono una soluzione ottima, ma producono sempre una soluzione ammissibile, in generale buona, in un tempo molto veloce.

Osservazione 1.4. Se gli algoritmi greedy raggiungono una soluzione ottima, la raggiungono molto velocemente

Il problema 1.2 ammette un algoritmo che produce una soluzione ottima, ma in un tempo maggiore.

Algoritmo 1.1.2: Algoritmo ingenuo per il problema della vetta di altezza massima

```
1 foreach vette nel profilo montuoso do
2 |   passo alla vetta successiva ricordandomi la più alta su cui sono
  |     passato;
3 ritorno indietro alla più alta visitata.
```

Ipotizzando di partire dal centro di un profilo montuoso di n vette, con l'algoritmo ingenuo servono, nel caso pessimo in cui la vetta massima sia all'estrema destra, $\frac{n}{2}$ passi per arrivare all'estremo destro, n passi per attraversare il profilo montuoso arrivando all'estremo sinistro, e altri n passi per ritornare all'estremo destro. Il numero totale di passi effettuati dall'alpinista è

$$\frac{n}{2} + n + n = \frac{5}{2}n$$

Con l'algoritmo 1, dato che non si retrocede, si arriva ad una soluzione ammissibile non ulteriormente migliorabile in $\frac{n}{2}$ passi. L'algoritmo greedy è quindi 5 volte più veloce dell'algoritmo ingenuo.

1.2 Algoritmo del resto

Uno dei problemi più semplici che si risolve con un metodo greedy è il *problema del resto*.

Problema 1.5. Dato un prezzo e una quantità pagata, determinare il resto utilizzando il numero minimo di monete o banconote.

L'insieme dell soluzioni ammissibili K è l'insieme di tutte le combinazioni di tagli il cui valore è uguale al resto, mentre la funzione obiettivo f è quella che associa ad ogni insieme di quantità di tagli la sua cardinalità. Un approccio greedy per risolvere questo problema è quello di partire dal taglio più grande e calcolare quante banconote o monete ci stanno nella soluzione ottima. Con il resto rimanente si procede in modo analogo.

Algoritmo 1.2.1: Algoritmo greedy per il problema del resto

```
1 foreach taglio di monete/banconote ordinate in ordine decrescente
    do
2 |   calcolo quante monete/banconote ci stanno nel resto;
```

Il problema 1.5 ammette un algoritmo greedy che produce una soluzione ottima.

1.3 Algoritmo del Knapsack

Problema 1.6. Dato uno zaino di peso massimo $P > 0$ e un insieme $T = \{1, \ldots, n\}$ di *tipi* di oggetti di cui si conoscono i pesi (positivi) $p_1, \ldots, p_n$ e i valori (positivi) $v_1, \ldots, v_n$, vogliamo riempire lo zaino non superando il peso massimo e massimizzando il valore degli elementi inseriti. Gli oggetti sono presenti con disponibilità infinita.

Se gli oggetti sono di molti tipi e la capacità dello zaino è molto grande, le possibilità diventano troppe. Si può procedere in diversi modi per risolvere questo problema:

1. Inserisco nello zaino gli elementi in ordine decrescente di valore (punto sulla qualità);

2. Inserisco nello zaino gli elementi in ordine crescente di peso (punto sulla quantità);

3. Inserisco nello zaino gli elementi in ordine decrescente di rapporto valore-peso (cerco un compromesso);

Il primo metodo trova facilmente un controesempio. Ipotiziamo di avere a disposizione uno zaino di capacità 10 e due tipologie di oggetti i cui pesi e valori sono rispettivamente:

$$p = (8, 3) \quad v = (4, 2) \tag{1.1}$$

La soluzione ottenuta dal primo algoritmo è quella che inserisce nello zaino solo il primo elemento di peso 8 e valore 4, anche se la soluzione ottima è quella che inserisce nello zaino tre copie dell'elemento 2, raggiungendo un valore totale di 6. Il secondo metodo risolve correttamente l'istanza precedente, ma fallisce se analiziamo gli oggetti di peso e valore:

$$p = (8, 3) \quad v = (16, 3) \tag{1.2}$$

La soluzione ottenuta dal secondo algoritmo è quella che inserisce nello zaino tre copie dell'oggetto due, totalizzando un valore di 9, mentre la soluzione ottima è quella che inserisce nello zaino il primo oggetto, raggiungendo un valore di 16. Il terzo metodo notiamo che funziona in tutti e due i casi. Nell'esempio 1.1 i rapporti valore-peso sono:

$$\frac{v}{p} = \left(\frac{4}{8}, \frac{2}{3}\right)$$

e dato che il maggiore è $0.\overline{6}$, vengono inserite nello zaino tre copie dell'elemento 2. Nell'esempio 1.2 i rapporti valore-peso sono:

$$\frac{v}{p} = \left(\frac{16}{8}, \frac{3}{3}\right)$$

e dato che il maggiore è 2, vengono inserite nello zaino tre copie dell'elemento 2. È stato dimostrato nel 1990 da Martello e Toth che questo metodo arriva ad una soluzione ottima. (Silvano Martello, Paolo Toth, Knapsack Problems: Algorithms and Computer Implementations). Questo è uno dei problemi più importanti dell'ottimizzazione matematica in quanto ammette diverse varianti con moltissime applicazioni pratiche. Possiamo quindi formulare un algoritmo greedy funzionante per risolvere il problema 1.6

Algoritmo 1.3.1: Algoritmo greedy per il problema del knapsack

```
1 foreach i ∈ T do
2 |  calcolo A_i ← v_i / p_i;
3 foreach A_i ordinati in modo decrescente do
4 |  se l'elemento i ci sta nello zaino ne inserisco più copie possibili.
```

1.4 Schedulazione delle attività

Un altro problema che si può risolvere con un approccio greedy è quello della schedulazione delle attività.

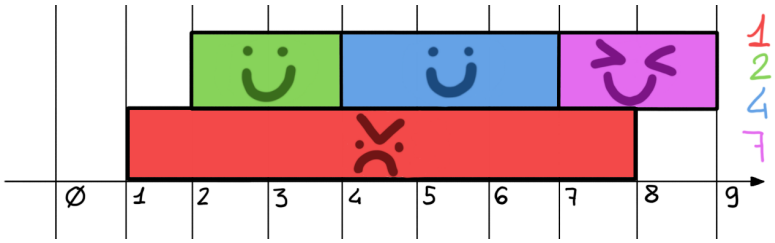
Problema 1.7. Dato un insieme di attività $T = \{1, \ldots, n\}$ con dei relativi istanti di inizio $S = \{s_1, \ldots, s_n\}$ e di fine $F = \{f_1, \ldots, f_n\}$ con $s_i < f_i$, determinare un insieme di cardinalità massima di attività compatibili. Due attività i e j si dicono compatibili se $f_i \leq s_j$ oppure $f_j \leq s_i$.

I criteri greedy con cui scegliere le attività possono essere:

- *Earliest start time*: prendo le attività in ordine crescente rispetto gli istanti iniziali;
- *Shortest interval*: prendo le attività più brevi;
- *Fewest conflicts*: prendo le attività in ordine crescente del numero di conflitti che generano;
- *Earliest finish time*: prendo le attività in ordine crescente rispetto gli istanti finali;

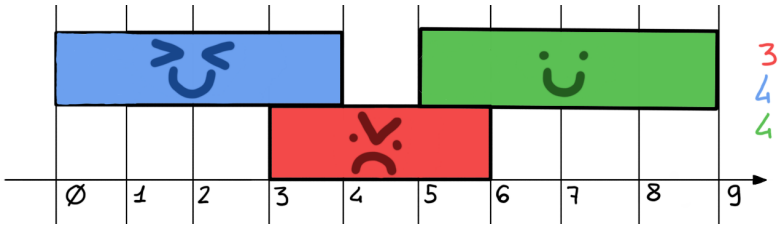
Con il metodo *Earliest start time* può capitare che un'attività che comincia per prima finisca dopo molte altre, come in figura

Figura 1.5: Earliest start time.



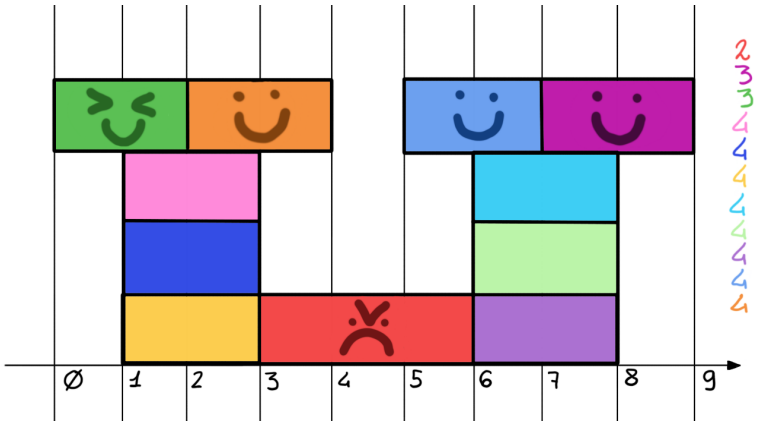
Con il metodo *Shortest interval* può capitare che un'attività che un'attività che dura meno di altre due le renda non compatibili, come in figura

Figura 1.6: Shortest interval.



Con il metodo *Fewest conflicts* può capitare che un'attività che un'attività che genera pochi conflitti renda inutilizzabili molte attività che ne generano di più.

Figura 1.7: Fewest conflicts.



La soluzione migliore è adottare una politica *earliest finish time*. Possiamo quindi definire un algoritmo greedy per risolvere li problema 1.7.

Algoritmo 1.4.1: Algoritmo greedy per il problema della schedulazione delle attività

```
1 Ordino le attività per ordine crescente di istante finale;
2 foreach i ∈ T do
3     se l'attività i è compatibile con quelle nella soluzione la aggiungo;
```

Capitolo 2

La ricorsione e la programmazione dinamica

Definizione 2.1. Un algoritmo si dice ricorsivo se ha bisogno di se stesso per essere definito.

Algoritmo 2.0.1: Esempio di algoritmo ricorsivo

```
1 Algoritmo: f(x)
2 return 1+2f(x);
```

Se proviamo a calcolare questo particolare esempio per qualunque valore di x la computazione diverge, perché non ci sono casi in cui l'algoritmo possa essere definito in modo autonomo. Per ovviare a questo problema, gli algoritmi ricorsivi devono avere almeno un caso base, ossia una configurazione dei parametri di input che non necessitano dell'algoritmo stesso per essere processati. Un esempio banale è il calcolo del fattoriale. Una prima (ed errata) implementazione può essere:

Algoritmo 2.0.2: Fattoriale errato

```
1 Algoritmo: f(x)
2 return x· f(x − 1);
```

La computazione di questo algoritmo non termina mai. Una più corretta implementazione è:

Algoritmo 2.0.3: Fattoriale corretto

```
1 Algoritmo: f(x)
2 if x = 0 then
3 |  return 1;
4 else
5 |  return x· f(x − 1);
```

Per progettare un algoritmo ricorsivo dobbiamo tenere quindi a mente due condizioni necessarie:

1. l'algoritmo deve avere un *caso base* che non contiene chiamate ricorsive;
2. nelle chiamate successive i parametri devono tendere verso il caso base.

Osservazione 2.2. L'idea chiave per attuare un ragionamento ricorsivo per risolvere un problema è quella di ridurre un'istanza *grande* di un problema a un'istanza *più piccola* **dello stesso problema**, fino ad arrivare ad un caso banale.

Possiamo definire i principali operatori $(+, -, \times, /)$, per semplicità con parametri interi positivi x e y con $x \geq y$, mediante degli algoritmi ricorsivi:

Algoritmo 2.0.4: Somma ricorsiva

```
1 Algoritmo: somma(x)
2 if y = 0 then
3 |  return x;
4 else
5 |  return somma(x + 1,y − 1);
```

Algoritmo 2.0.5: Sottrazione ricorsiva

```
1 Algoritmo: sottrazione(x)
2 if y = 0 then
3 |  return x;
4 else
5 |  return sottrazione(x − 1,y − 1);
```

Algoritmo 2.0.6: Moltiplicazione ricorsiva

```
1 Algoritmo: moltiplicazione(x)
2 if y = 0 then
3 |  return 0;
4 else
5 |  return x+moltiplicazione(x,y − 1);
```

Algoritmo 2.0.7: Divisione intera ricorsiva

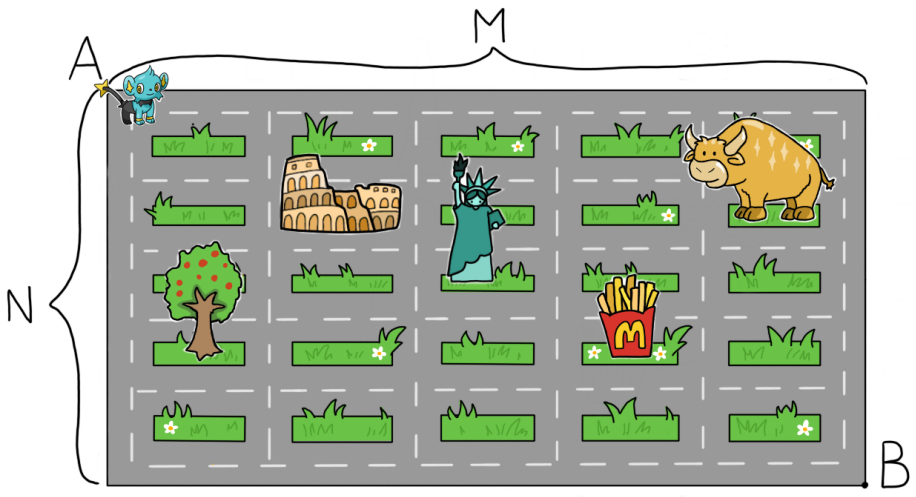
```
1 Algoritmo: divisioneIntera(x)
2 if x < y then
3 |   return 0;
4 else
5 |   return 1+divisioneIntera(x − y,y);
```

2.1 I percorsi di Manhattan

Supponiamo di trovarci a Manhattan, isola americana nota per la particolare rete stradale. È composta da strade orizzontali, dette *street* e strade verticali dette *avenue*.

Problema 2.3 (Manhattan)**.** Supponendo di muoverci solo a destra o in basso, conoscendo il numero di street N e di avenue M, vogliamo calcolare quanti sono i possibili percorsi che collegano l'estremo nord-ovest con l'estremo sud-est.

Figura 2.1: Percorsi di Manhattan.



Un metodo di ragionare è procedere per *backtracking*. Cerco di muovermi solo a destra, quando non posso più cerco di muovermi in basso e ripeto, se non riesco più a procedere a destra o in basso retrocedo fino a quando posso compiere nuove mosse. Questa tecnica, efficiente per molti problemi come quello delle otto regine, è altamente inefficiente per questo problema

perché calcola effettivamente tutti i percorsi, mentre è chiesto solamente di determinarne il numero. Per realizzare un algoritmo ricorsivo occorre prima di tutto individuare i casi base, ossia i casi banali. Per il problema 2.3 sono:

- quelli in cui l'isola è composta da una sola street (in cui c'è un unico percorso che va dall'estremo ovest all'estremo est);
- quelli in cui l'isola è composta da una sola avenue (in cui c'è un unico percorso che va dall'estremo nord all'estremo sud).

Nell'algoritmo il caso base sarà quidi:

Algoritmo 2.1.1: Passo base dei percorsi di Manhattan

```
1 Algoritmo: Manhattan(N, M)
2 if N = 1 ∨ M = 1 then
3 |  return 1;
```

Per il caso ricorsivo ragioniamo nel seguente modo:

- se mi muovo a destra i percorsi sono quelli di una Manhattan con lo stesso numero di street ma una avenue in meno;
- se mi muovo in basso i percorsi sono quelli di una Manhattan con lo stesso numero di avenue ma una street in meno;

Il numero di percorsi totali sarà la somma di questi due. Otteniamo quindi un algoritmo che risolve il problema 2.3.

Algoritmo 2.1.2: Calcolo dei percorsi di Manhattan

```
1 Algoritmo: Manhattan(N, M)
2 if N = 1 ∨ M = 1 then
3 |  return 1;
4 else
5 |  return Manhattan(N,M-1) + Manhattan(N-1,M);
```

2.2 Memoization

Non tutti questi algoritmi non sono efficienti da soli perché effettuano calcoli molto complessi molte volte, ma, con la tecnica della *memoization*, possiamo salvare le computazioni effettuate per evitare di svolgere nuovamente gli stessi calcoli. Consideriamo il noto problema di Fibonacci.

Problema 2.4 (Fibonacci)**.** Dato un numero $n > 0$, calcolare l'n-esimo termine della sequenza di Fibonacci.

$$f(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ f(n-1) + f(n-2) & \text{altrimenti} \end{cases}$$

Un algoritmo che risolve questo problema è:

Algoritmo 2.2.1: Calcolo deli numeri di Fibonacci

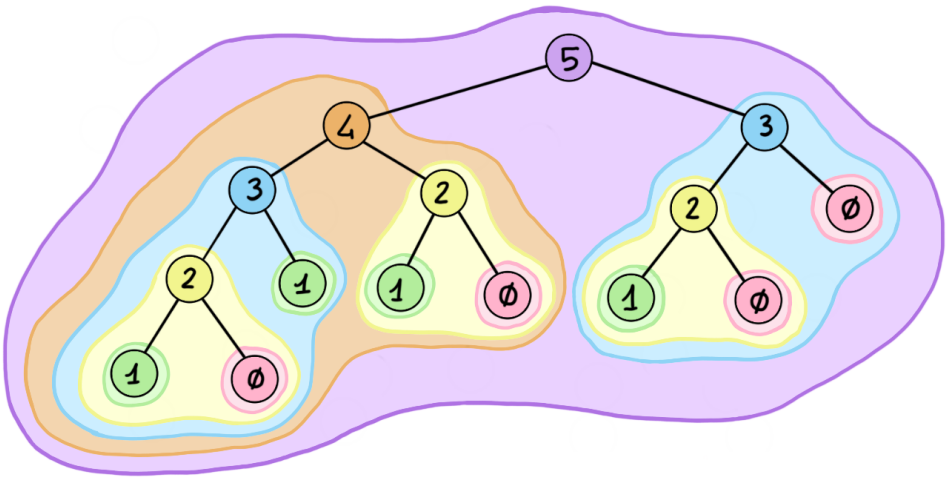
```
1 Algoritmo: Fib(n)
2 if n = 0 then
3 |   return 0;
4 else
5 |   if n = 1 then
6 |   |   return 1;
7 |   else
8 |   |   return Fib(n − 1)+Fib(n − 2);
```

Una prima implementazione dell'algoritmo 8 produce la seguente funzione

```
1 int Fib(int n){
2   if(n == 0) return 0;
3   else if(n == 1) return 1;
4   else return Fib(n-1)+ Fib(n-2);
5 }
```

Con una semplice implementazione ci accorgiamo che il tempo di computazione esplode già con input molto bassi ($n = 50$).

Figura 2.2: Chiamate ricorsive di Fib.



Proviamo ora ad applicare la tecnica della *memoization* per l'algoritmo 8, ossia salviamoci i valori già calcolati in un array in modo che

$$\texttt{T[n]} = f(n)$$

```
int FibMem(int n){
if(T[n] == -1)
  if(n == 0) T[0] = 0;
  else if(n == 1) T[1] = 1;
  else T[n] = FibMem(n-2)+ FibMem(n-1);
  return T[n];
}
```

Notiamo che con questa tecnica il tempo di esecuzione diminuisce drasticamente, rendendo la computazione decisamente più gestibile. Non sempre la ricorsione necessita della memoization, ad esempio per il calcolo del fattoriale il numero di chiamate ricorsive sono uguali nei due casi, ma in molti problemi è utile salvare i risultati parziali.

Osservazione 2.5. La ricorsione ha un approccio *top-down*: parte da un'istanza grande di un problema e cerca di ridurlo in istanze più piccole e semplici da risolvere, fino a dei casi banali.

2.3 Rod-cut problem

Consideriamo il *rod-cut problem* (RCP):

Problema 2.6 (rod-cut problem)**.** Data una barra di lunghezza n e n prezzi $p_1, \ldots, p_n$, determinare il profitto massimo che si può ottenere tagliando la barra in pezzi, sapendo che ogni pezzo di lunghezza l genera un profitto p_l.

Un pezzo di lunghezza n può essere tagliato in 2^{n-1} modi diversi. Prendiamo una soluzione ottima dove tagliamo la barra di lunghezza n in $1 \leq k \leq n$ parti di lunghezza $i_1, \ldots, i_k$ con $n = i_1 + \cdots + i_k$. Sicuramente se $n = 1$, allora $k = 1$, $i_1 = 1$ e il valore ottimo è p_1. Se invece $n = 2$, allora sono possibili due casi:

- tagliare la barra ottenendo un profitto di $2p_1$;
- tagliare la barra a metà ottenendo un profitto di p_2.

Prendiamo come valore ottimo del problema con la barra di lunghezza $j < n$ il valore r_j. Possiamo definire la soluzione generale dell'istanza di parametro n il valore:

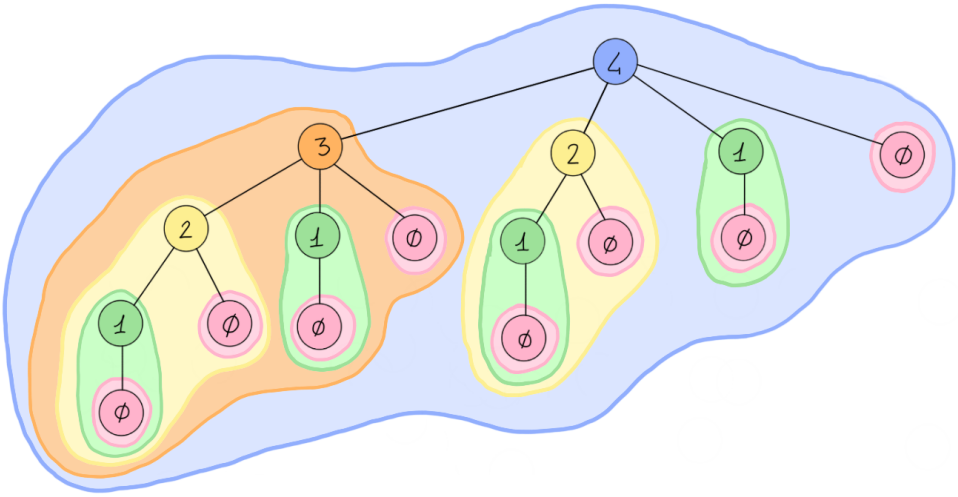
$$r_n = \max\{p_n, (r_1 + r_{n-1}), (r_2 + r_{n-2}), \ldots, (r_{n-1} + r_1)\}.$$

Il seguente algoritmo riceve in input un insieme di valori p di prezzi e un intero $n > 0$ e manda in output r_n.

Algoritmo 2.3.1: Algoritmo ricorsivo per risolvere il problema 2.6

```
1 Algoritmo: RC(n)
2 if n = 0 then
3     return 0;
4 q=-∞;
5 for i ∈ {1, ..., n} do
6     q ← max {q, p_i+RC(n − i)};
7 return q;
```

Figura 2.3: Chiamate ricorsive di RC.



Notiamo subito che l'albero delle chiamate ricorsive ha 2^{n-2} e che molte istanze vengono ripetute più volte. Invece di risolvere un sottoproblema più volte è utile risolverlo una sola volta e salvarsi la soluzione. Possiamo applicare anche a questo algoritmo la memoization migliorando il tempo di esecuzione.

```
int RCMem(int n){
  if(T[n] == -1){
    if(n == 0){
      T[n] = 0;
    }else{
      int q = -1;
      for(int i = 1 ; i <= n ; i++){
        q = max(q,p[i]+RCMem(n-i));
      }
      T[n] = q;
    }
  }
  return T[n];
}
```

2.4 Programmazione dinamica

Un approccio simmetrico rispetto alla ricorsione è quello *bottom-up* della *programmazione dinamica*: Si risolvono prima i casi banali e con quelli si cerca di costruire soluzioni semplici di problemi complessi. È tipicamente

(ma non solo) utilizzata per problemi di ottimizzazione definiti in 1.1. Consideriamo il problema 2.4. Al posto di partire a costruire il risultato finale cerchiamo di espanderci dai dati iniziali. Sapendo che $f(0) = 0$ e $f(1) = 1$ posso calcolare

$$f(2) = f(0) + f(1) = 0 + 1 = 1$$

e procedendo così possiamo calcolare i termini successivi fermandoci quando arriviamo a n.

```
int fibonacciDinProg(int n){
  if(n == 0) return 0;
  else if(n == 1) return 1;
  else{
    int a = 0;
    int b = 1;
    for(int i = 2 ; i <= n ; i++){
      int c = a + b;
      a = b;
      b = c;
    }
    return b;
  }
}
```

La programmazione dinamica richiede spesso una struttura dati di appoggio per la computazione. Quindi questa tecnica va usata assieme alla memoization. A causa della struttura dei ragionamenti, in alcuni casi è più utile utilizzare un approccio iterativo piuttosto che ricorsivo. Per il problema 2.6 possiamo introdurre una struttura dati analoga a quella utilizzata nell'algoritmo con la memoization. Ragioniamo nel seguente modo:

- Se $n = 1$ c'è solo un modo di tagliare la barra;
- Se $n > 1$ suppongo di aver già calcolato tutti i valori ottimi per le barre di lunghezza $m < n$. Il valore ottimo per la barra di lunghezza n è ottenuto in modo analogo al caso ricorsivo controllando tutte le combinazioni possibili.

Algoritmo 2.4.1: Algoritmo di programmazione dinamica per risolvere il problema 2.6

```
1 Algoritmo: Fib(n)
2 for j = 1, ..., n do
3 |   q ← −∞;
4 |   for i = 1, ..., j do
5 |   |   q ← max {q, p_i + r_{j−i}};
6 |   r_j ← q;
7 return r[n];
```

2.5 Algoritmo del knapsack

Consideriamo di nuovo il problema 1.6. La correttezza dell'algoritmo che studieremo si basa principalmente sulla seguente osservazione:

Osservazione 2.7. Se da una soluzione ottima per lo zaino di dimensione P tolgo un oggetto t, ottengo una soluzione ottima per il problema di dimensione $P - p_t$.

Conoscendo le soluzioni ottime del problema di dimensione P, possiamo conoscere le soluzioni ottime del problema $P + t$ per ogni $t \in T$. Se il valore della soluzione ottima per un problema di dimensione P ha valore V, allora esiste anche una soluzione ottima di valore $V + v_t$ per un problema di dimensione $P + p_t$. Tutte le soluzioni ottime sono ottenute in questo modo. Costruisco un vettore R di lunghezza P che costruisce le soluzioni ottime delle istanze di dimensioni minori. Alla fine la soluzione sarà in R_P. Una volta appurato che nello zaino c'è spazio per l'n-esimo oggetto, decido se è più conveniente inserirlo o lasciare lo spazio per altri oggetti.

Algoritmo 2.5.1: Algoritmo ricorsivo per risolvere il problema 1.6

```
1 Algoritmo: KP(P, n)
2 if n = 0 ∨ P = 0 then
3 |   return 0;
4 if p_n > P then
5 |   return KP(P, n − 1);
6 else
7 |   return max {v_n+KP(P − p_n, n − 1),KP(P, n − 1)};
```

Questo algoritmo ricorsivo continua ad avere le solite problematiche discusse, per questo conviene memorizzarsi i risultati già calcolati in una struttura dati apposita. Usiamo una matrice M per memorizzare i dati delle

computazioni precedenti. $M_{i,j}$ contiene il valore ottimo del problema con i primi i oggetti e con uno zaino di dimensione j. L'algoritmo riceve in input il peso massimo P, il peso p, i valori v, il numero di elementi da inserire n e una matrice $M \equiv -\infty$ e manda in output il valore ottimo V.

Algoritmo 2.5.2: Algoritmo ricorsivo con memoization per risolvere il problema 1.6

```
1  Algoritmo: KP-M(P, n)
2  if n < 0 then
3      return 0;
4  if M_{n,P} ≠ −∞ then
5      return M_{n,P};
6  if p_n > P then
7      M_{n,P} ←KP-M(P, n − 1);
8      return M_{n,P};
9  else
10     M_{n,P} ← max {v_n+KP-M(P − p_n, n − 1),KP-M(P, n − 1)};
11     return M_{n,P};
```

Come per l'esempio precedente si può modifivare questo algoritmo risolvendolo bottom up mediante la programmazione dinamica. L'algoritmo riceve in input il peso massimo P, il peso p, i valori v e il numero di elementi da inserire n e manda in output il valore ottimo V.

Algoritmo 2.5.3: Algoritmo di programmazione dimanica per risolvere il problema 1.6

```
1  Algoritmo: KP-D(P, n)
2  M_{0,1}, . . . , M_{0,P} := 0;
3  for i = 1, . . . , n do
4      for w = 0, . . . , P do
5          if p_i ≤ w then
6              M_{i,w} ← max {M_{i−1,w}, v_i + M_{i−1,w−p_i}};
7          else
8              M_{i,w} ← M_{i−1,w};
9  return M_{n,P};
```

Capitolo 3

Teoria dei grafi

Definizione 3.1. Un *grafo orientato non pesato* $\mathcal{G}$ è definito da un insieme V detto insieme dei *nodi* con n elementi e un insieme $E \subseteq V \times V$ detto insieme degli *archi* con m elementi. Dato un arco $e = (i, j) \in E$, il nodo i è detto *sorgente* dell'arco e e il nodo j è detto *destinazione* dell'arco e.

I grafi sono molto utili per risolvere diversi problemi e possono essere visualizzati graficamente.

Esempio 3.2. Un esempio di grafo può essere:

$$\begin{aligned}\mathcal{G} := (&V = \{0, 1, 2, 3, 4\},\\ &E = \{(0, 1), (1, 3), (2, 0), (2, 3), (3, 4), (4, 2)\})\end{aligned}$$

e la sua rappresentazione grafica è:

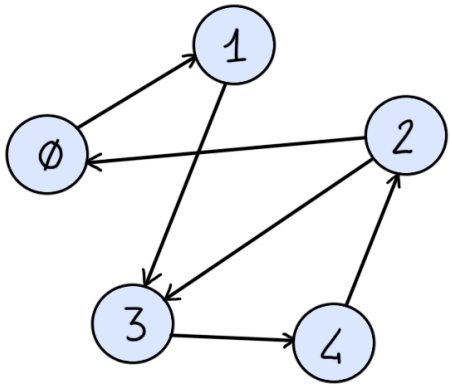


Figura 3.1: Esempio di grafo.

Definizione 3.3. Dato un grafo $\mathcal{G} = (V, E)$ diciamo che $e_1, \ldots, e_k$ è un *cammino* che collega v e w se il nodo sorgente di e_1 è v, il nodo destinazione di e_k è w e per ogni $l \in \{1, \ldots, k-1\}$ vale che il nodo destinazione di e_l è uguale al nodo sorgente di e_{l+1}. Un grafo è *connesso* se per ogni coppia di nodi esiste almeno un cammino che li collega. Un *circuito* è un cammino che parte e arriva nello stesso nodo.

3.1 Strutture dati per i grafi

Un primo metodo per la codifica di un grafo è quello della *matrice di adiacenza*.

Definizione 3.4. Dato grafo orientato e non pesato $\mathcal{G} = (V, E)$ con n nodi e m archi è possibile definire la sua *matrice di adiacenza* $A \in \mathcal{M}(n \times n, \{0, 1\})$ come segue:

$$A_{i,j} = \begin{cases} 1 & \text{se } (i,j) \in E \\ 0 & \text{altrimenti} \end{cases}$$

Ad un grafo $\mathcal{G}$ associamo quindi una matrice quadrata che ha tante righe e tante colonne quanti sono i nodi del grafo e nella posizione (i, j) ha valore 1 solo se esiste un arco da i a j. La matrice di adiacenza del grafo nell'esempio 3.2 è

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Una seconda possibilità per rappresentare un grafo è quella della *matrice di incidenza*.

Definizione 3.5. Dato grafo orientato e non pesato $\mathcal{G} = (V, E)$ con n nodi e m archi è possibile definire la sua *matrice di incidenza* $A \in \mathcal{M}(n \times m, \{0, \pm 1\})$ come segue:

$$B_{i,e} = \begin{cases} 1 & \text{se l'arco } e \text{ esce da } i \\ -1 & \text{se l'arco } e \text{ entra in } i \\ 0 & \text{altrimenti} \end{cases}$$

Ad un grafo $\mathcal{G}$ associamo quindi una matrice che ha tante righe quanti sono i nodi del grafo, tante colonne quanti sono gli archi del grafo e nella

posizione (i, e) ha valore ± 1 solo se l'arco e entra o esce dal nodo i. La matrice di adiacenza del grafo nell'esempio 3.2 è

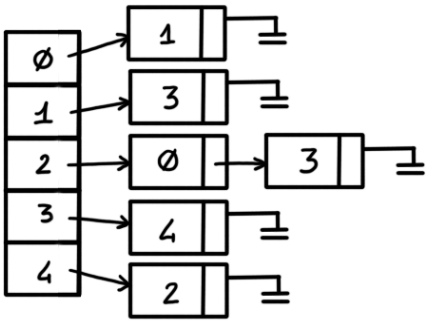
$$\begin{pmatrix} 1 & 0 & -1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & -1 \\ 0 & -1 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 \end{pmatrix}$$

Queste due strutture sono le più ustate per lo studio della teoria dei grafi, anche se per i nostri scopi è più utile introdurre due nuove struttura dati.

Definizione 3.6. Dato grafo orientato e non pesato $\mathcal{G} = (V, E)$ con n nodi e m archi è possibile definire la sua *lista di adiacenza* $\mathcal{L}$ come segue: Per ogni nodo $i \in V$ definisco una lista $\mathcal{L}_i$ contenente i nodi che possono essere raggiunti da i.

La lista di adiacenza del grafo nell'esempio 3.2 è

Figura 3.2: Lista di adiacenza del grafo in figura 3.1.



3.2 Visite di un grafo

Vogliamo definire degli algoritmi che visitino tutti i nodi che possono essere raggiunti da una sorgente lungo gli archi. Tutti e due i metodi che vedremo di applicano sulla lista di adiacenza del grafo e necessitano di una seconda lista di appoggio.

L'idea degli algoritmi di visita di un grafo è quella di partire da un nodo v e tenere una lista dei nodi da visitare che all'inizio contiene solo v. Fino a quando ci sono elementi nella lista se ne preleva uno, se non è stato visitato si visita e si aggiungono i suoi vicini alla lista. La differenza tra i due metodi che vedremo sta nella gestione della lista.

- Se la lista viene gestita come una *pila* (LIFO), la visita è in ampiezza.

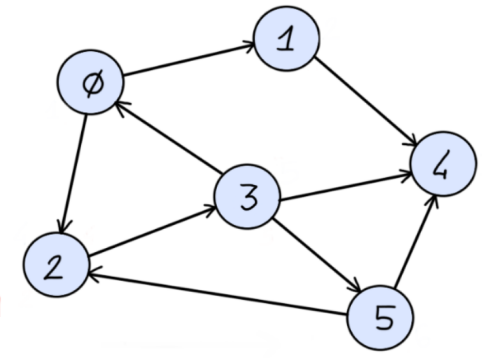
- Se la lista viene gestita come una *coda* (FIFO), la visita è in profondità.

Algoritmo 3.2.1: Visita di un grafo

```
1  Algoritmo: visita(v)
2  S = {v};
3  foreach nodi w del grafo do
4      V[w] ← 0;
5  while S ≠ ∅ do
6      Prelevo w da S;
7      if V[w] = 1 then
8          V[w] ← 1;
9          Visito w;
10         foreach nodo z adiacente a w do
11             Inserisco z in S;
```

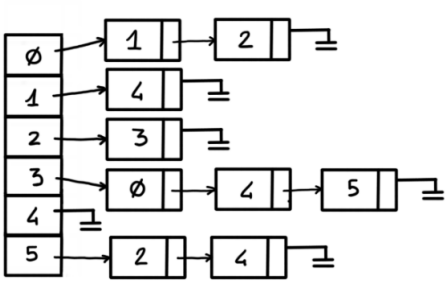
Esempio 3.7. Consideriamo il grafo $\mathcal{G}$ in figura:

Figura 3.3: Esempio di visite di un grafo.



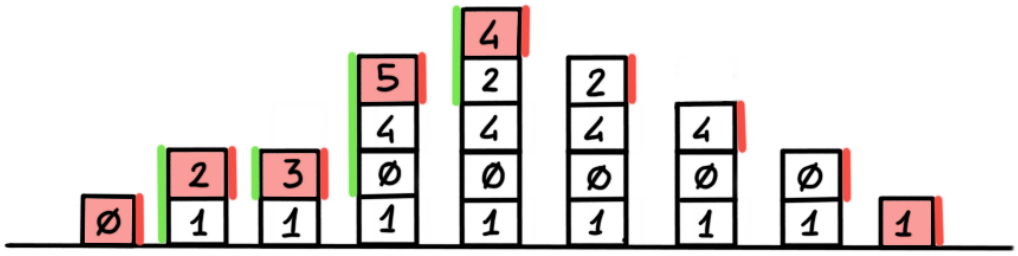
e la sua lista di adiacenza $\mathcal{L}$

Figura 3.4: Lista di adiacenza di $\mathcal{G}$.



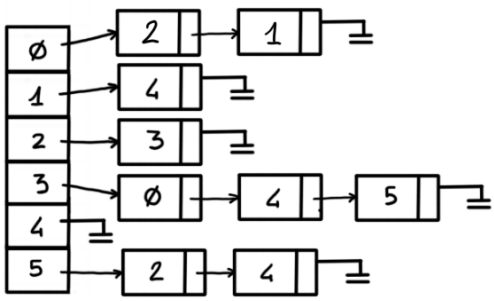
Applichiamo l'algoritmo 3.2.1 della visita in profondità, quindi con una gestione LIFO della lista dei nodi da visitare, partendo dal nodo 0.

Figura 3.5: Liste della visita in profondità di $\mathcal{G}$.



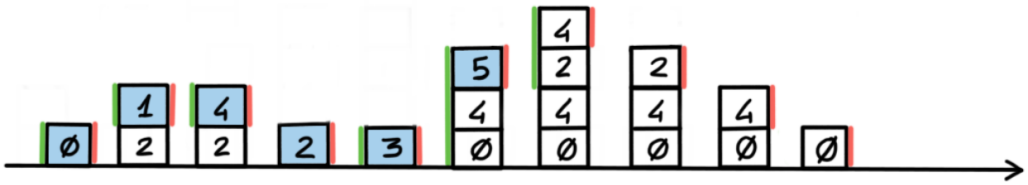
Notiamo che l'algoritmo visita tutti i nodi del grafo nell'ordine 0, 2, 3, 5, 4, 1. Il grafo in figura 3.3 può essere rappresentato anche dalla lista di adiacenza $\mathcal{L}'$

Figura 3.6: Lista di adiacenza alternativa di $\mathcal{G}$.



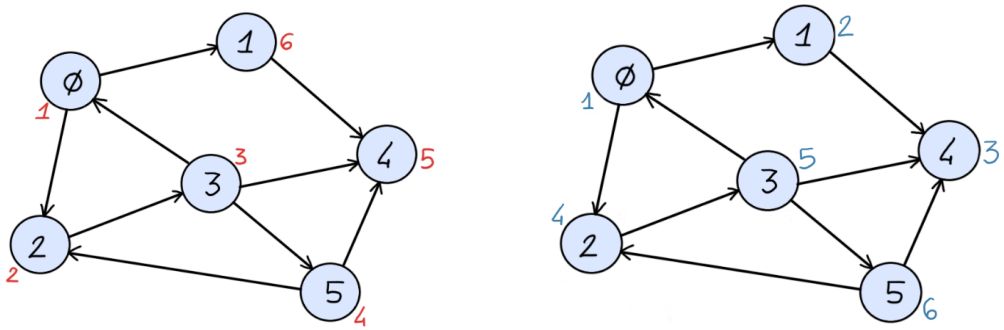
Applicando lo stesso algoritmo otteniamo una diversa lista dei nodi da visitare

Figura 3.7: Liste della visita in profondità alternativa di $\mathcal{G}$.



e quindi la sequenza 0, 1, 4, 2, 3, 5.

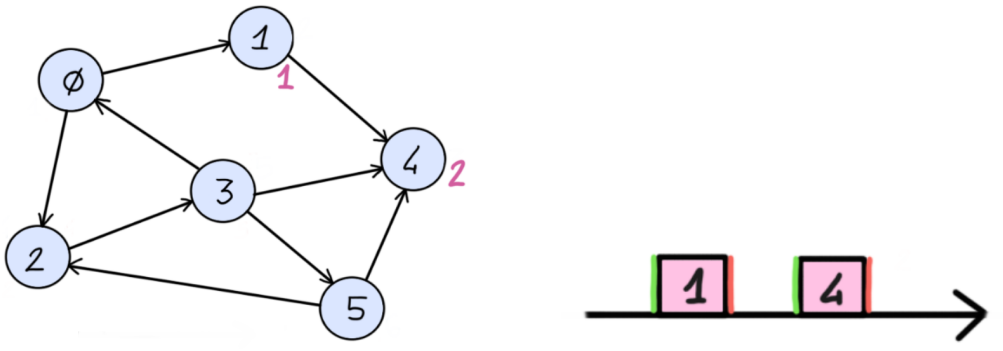
Figura 3.8: Visite in profondità di $\mathcal{G}$.



Le sequenze sono diverse, ma i nodi che si raggiungono sono uguali.

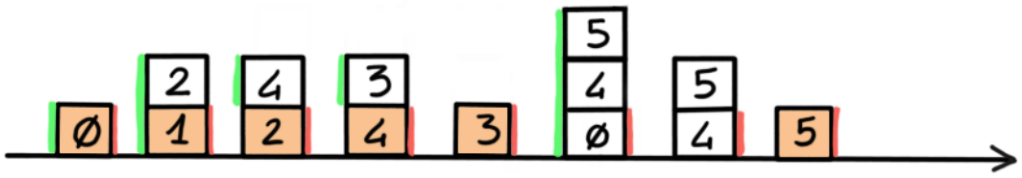
Non è detto che partendo da un qualunque nodo del grafo si riesca a ottenere la stessa sequenza, infatti, se nel grafo in figura 3.3 partiamo dal nodo 1, otteniamo la seguente lista dei nodi da visitare:

Figura 3.9: Visita in profondità di $\mathcal{G}$ partendo dal nodo 2.



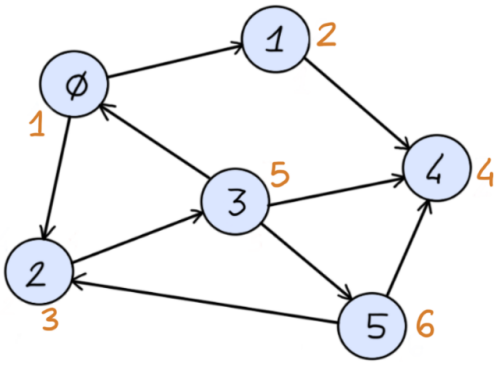
Applicando l'algoritmo di visita in ampiezza, quindi una gestione FIFO della lista, su $\mathcal{G}$ otteniamo la lista dei nodi visitati:

Figura 3.10: Liste della visita in ampiezza di $\mathcal{G}$.



otteniamo la sequenza dei nodi 0, 1, 2, 4, 3, 5. Notiamo che, a meno dell'ordine, i due algoritmi visitano gli stessi nodi.

Figura 3.11: Visita in ampiezza di $\mathcal{G}$.



Modificando leggermente l'algoritmo 3.2.1 possiamo determinare se un grafo è connesso.

Algoritmo 3.2.2: Verifica della connessione di un grafo

```
1 Algoritmo: Connessione(v)
2 foreach nodo w del grafo do
3 |  Verifico se riesco a visitare tutti i nodi del grafo tramite
  |   l'algoritmo 3.2.1;
```

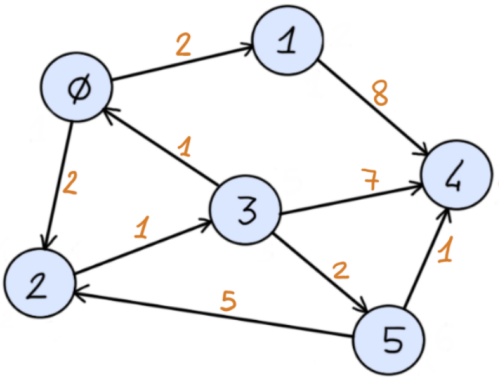
3.3 Algoritmo di Dijkstra

Per questa sezione dobbiamo definire una nuova famiglia di grafi.

Definizione 3.8. Un *grafo pesato* $\mathcal{G}$ è dato da un insieme di nodi V, un insieme di archi E e una funzione dei pesi $\psi : E \to \mathbb{Z}$ che associa ad ogni arco un peso.

Questo peso può essere interpretato come un costo per attraversare l'arco.

Figura 3.12: Esempio di un grafo pesato.



Esempio 3.9.

Questa informazione va a modificare le strutture dati che utilizzavamo per rappresentare i grafi non pesati.

Definizione 3.10. Dato grafo orientato e pesato $\mathcal{G} = (V, E, \psi)$ con n nodi e m archi è possibile definire la sua *matrice di adiacenza* $A \in \mathcal{M}(n \times n, \{0, 1\})$ come segue:

$$A_{i,j} = \begin{cases} \psi(i,j) & \text{se } (i,j) \in E \\ 0 & \text{altrimenti} \end{cases}$$

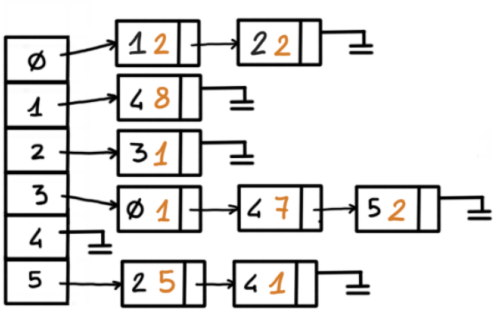
Ad un grafo pesato $\mathcal{G}$ associamo quindi una matrice quadrata che ha tante righe e tante colonne quanti sono i nodi del grafo e nella posizione (i, j) ha valore $\psi(i, j)$ solo se esiste un arco da i a j. La matrice di adiacenza del grafo nell'esempio 3.9 è

$$\begin{pmatrix} 0 & 2 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 7 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 1 & 0 \end{pmatrix}$$

Definizione 3.11. Dato grafo orientato e pesato $\mathcal{G} = (V, E, \psi)$ con n nodi e m archi è possibile definire la sua *lista di adiacenza* $\mathcal{L}$ come segue: Per ogni nodo $i \in V$ definisco una lista $\mathcal{L}_i$ contenente i nodi j che possono essere raggiunti da i e i relativi pesi $\psi(i, j)$.

La lista di adiacenza del grafo nell'esempio 3.9 è

Figura 3.13: Lista di adiacenza del grafo in figura 3.12.



L'algoritmo di Dijkstra richiede che il grafo abbia solo pesi positivi, altrimenti non è garantita la correttezza. Un algoritmo (greedy) computazionalmente peggiore, ma che richiede semplicemente che non esistano circuiti di costo negativo, è quello di *Bellman Ford*.

Consideriamo un grafo pesato $\mathcal{G} = (V, E, \psi)$ e un nodo $v_0 \in V$. Se il grafo è connesso esistono a priori più percorsi da v verso ogni altro nodo del grafo. In questa sezione descriveremo un algoritmo che determina un cammino di peso minimo che collega due nodi in un grafo.

Questo è uno dei problemi più studiati nella teoria dei grafi e il più famoso algoritmo che risolve questo problema è l'algoritmo di Dijkstra.

Questo algoritmo di programmazione dinamica utilizza quattro strutture dati di appoggio e una politica greedy che aiuta a garantire la correttezza.

- S è l'insieme dei nodi già analizzati.
- T è l'insieme dei nodi da analizzare.
- f è una funzione che associa ad ogni nodo la lunghezza del cammino di peso minimo che collega f a quel nodo.
- P è una funzione che associa ad ogni nodo il precedente nel cammino da v_0 al nodo stesso.

La correttezza di questo algoritmo è garantita dalla seguente osservazione:

Osservazione 3.12. Presi tre nodi v, w e z in un grafo, se γ_1 è il cammino di peso minimo che colega v a w e γ_2 è il cammino di peso minimo che collega w a z, allora $\gamma_1\gamma_2$ è il cammino di peso minimo che collega v a z.

Algoritmo 3.3.1: Algoritmo di Dijkstra

```
1  Algoritmo: Dijkstra(v_0)
2  S = {v_0};
3  T = V \ S;
4  f(v_0) = 0;
5  P(v_0) = v_0;
6  foreach w ∈ V diverso da v_0 do
7      if w è adiacente a v_0 then
8          f(w) = ψ(v_0, w);
9          P(w) = v_0;
10     else
11         f(w) = +∞;
12         P(w) = −1;
13 while T ≠ ∅ oppure f(v) = +∞ per ogni v ∈ T do
14     Rimuovo da T un vertice v con il valore minimo di f e lo
        inserisco in S;
15     if f(v) ≠ +∞ then
16         foreach nodo w adiacente a v do
17             a = f(v) + ψ(v, w);
18             if a < f(w) then
19                 f(w) = a;
20                 P(w) = v;
```

L'algoritmo è diviso in una fase di inizializzazione (righe 2-12) e in una fase di ricerca dei percorsi minimi (righe 13-20).

2-3 Inserisco v_0 in S e tutti gli altri nodi in T;

4-5 Indico che v_0 dista 0 da se stesso ed è se stesso il suo predecessore.

6-12 Per ogni altro nodo adiacente a v_0 indico che ha distanza 1 da v_0 ed il suo predecessore è v_0. Per ogni altro nodo non adiacente a v_0 indico che ha distanza infinita da v_0 ed il suo predecessore non è definito.

13 Finche ci sono ancora nodi da anlizzare o quelli ancora da analizzare non dono raggiungibili.

14 Prelevo un nodo v da T con il valore minimo di f e lo inserisco in S.

15 Se il nodo considerato è raggiungibile.

16-17 Scorro i nodi w adiacenti a v e per ciascuno di questi calcolo la lunghezza del cammino $v_0 \to v \to w$.

18-20 Se questo valore è minore del cammino più breve che arriva in w calcolato fino ad ora, lo aggiorno.

Esempio 3.13. Proviamo ad applicare l'algoritmo 3.3.1 al grafo in figura 3.3 partendo dal nodo 0. Dopo l'inizializzazione otteniamo:

- $S = \{0\}$.
- $T = \{1, 2, 3, 4, 5\}$.
- $f = (0_0, 2_1, 2_2, +\infty_3, +\infty_4, +\infty_5)$.
- $P = (0_0, 0_1, 0_2, -1_3, -1_4, -1_5)$.

Per cercare i percorsi minimi l'algoritmo compie 5 iterazioni

1. Il nodo con il valore minimo di f è il nodo 1 (si può prendere anche il nodo 2). $S = \{0, 1\}$, $T = \{2, 3, 4, 5\}$. Dal nodo 1 si raggiunge il nodo 4 con peso 8, quindi

$$f = (0_0, 2_1, 2_2, +\infty_3, 1 + 8 = 9_4, +\infty_5) \quad P = (0_0, 0_1, 0_2, -1_3, 1_4, -1_5)$$

2. Il nodo con il valore minimo di f è il nodo 2. $S = \{0, 1, 2\}$, $T = \{3, 4, 5\}$. Dal nodo 2 si raggiunge il nodo 3 con peso 1, quindi

$$f = (0_0, 2_1, 2_2, 2 + 1 = 3_3, 2_4, +\infty_5) \quad P = (0_0, 0_1, 0_2, 2_3, 1_4, -1_5)$$

3. Il nodo con il valore minimo di f è il nodo 3. $S = \{0, 1, 2, 3\}$, $T = \{4, 5\}$. Dal nodo 3 si raggiungono i nodi 0, 4 e 5, ma solo per il nodo 5 si trova un cammino migliore che termina con un arco di peso 2, quindi

$$f = (0_0, 1_1, 1_2, 2_3, 2_4, 3 + 2 = 5_5) \quad P = (0_0, 0_1, 0_2, 2_3, 1_4, 3_5)$$

4. Il nodo con il valore minimo di f è il nodo 4. $S = \{0, 1, 2, 3, 4\}$, $T = \{5\}$. Dal nodo 4 non si raggiungono altri nodi, quindi f e P restano invariate.

5. L'ultimo nodo rimanente è il 5 $S = \{0, 1, 2, 3, 4, 5\}$, $T = \emptyset$. Dal nodo 5 si raggiunge il nodo 2 e 4, ma solo verso il nodo 4 si trova un cammino migliore che termina con un arco di peso 1, quindi

$$f = (0_0, 1_1, 1_2, 2_3, 5 + 1 = 6_4, 3_5) \quad P = (0_0, 0_1, 0_2, 2_3, 5_4, 3_5)$$

L'algoritmo termina perchè $T = \emptyset$ indicando che i percorsi minimi da 0 verso gli altri nodi sono lunghi in ordine 1, 1, 2, 2 e 3.

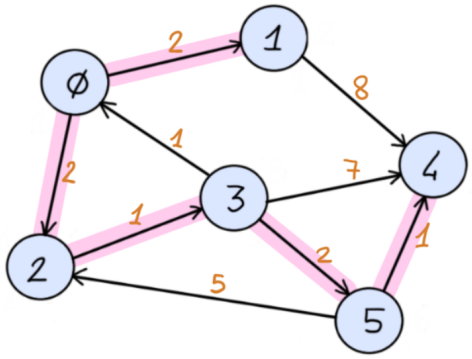


Figura 3.14: Cammini minimi nel grafo 3.9.

Con la struttura P possiamo ricostruire a ritroso il cammino da v a v_0, mediante il seguente algoritmo.

Algoritmo 3.3.2: Percorso minimo calcolato da 3.3.1

```
1 Algoritmo: camminoMinimo(v)
2 w ← v;
3 s ← w;
4 do
5 |   w ← P(w);
6 |   s ← (w → "s");
7 while w ≠ P(w);
```

Esempio 3.14. Se nell'esempio di 3.13 applichiamo l'algoritmo 3.3.2 sul nodo 5 otteniamo per ogni iterazione:

1. $w = 5 \to 3$ e $s = 3 \to 5$. Dato che $w = 3 \neq P(w) = 2$ itero;
2. $w = 3 \to 2$ e $s = 2 \to 3 \to 5$. Dato che $w = 2 \neq P(w) = 1$ itero;
3. $w = 2 \to 1$ e $s = 1 \to 2 \to 3 \to 5$. Dato che $w = 1 = P(w) = 0$ itero;
4. $w = 1 \to 0$ e $s = 0 \to 1 \to 2 \to 3 \to 5$. Dato che $w = 0 = P(w) = 0$ esco dal ciclo

Nota bene 3.15. L'algoritmo 3.3.1 può essere classificato come algoritmo di programmazione dinamica, infatti sfrutta la struttura dati f riempiendola in modo efficiente e utilizzando sempre valori precedentemente calcolati.

3.4 Grafi Euleriani e Hamiltoniani

In questa sezione considereremo un nuovo tipo di grafo.

Definizione 3.16. Un *grafo non orientato* $\mathcal{G}$ è definito da un insieme di nodi V e un insieme di coppie non ordinate di nodi E detto insieme di archi.

Ad ogni nodo di un grafo non orientato è possibile attribuire un valore.

Definizione 3.17. Dato un grafo non orientato $\mathcal{G} = (V, E)$, possiamo definire il *grado* di un nodo $v \in V$ come il numero di archi $\mathrm{d}(v)$ che incidono in quel nodo.

Tutti i grafi che considereremo in questa sezione saranno non orientati.
Immaginiamo di trovarci a Konigsberg

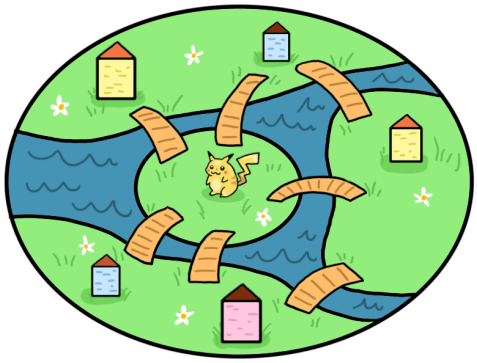


Figura 3.15: Ponti di Konigsberg.

e ci chiediamo se è possibile visitare tutta la città attraversando ogni ponte esattamente una volta. Uno dei primi matematici che ha affrontato questo problema è Eulero nel 1735.

Definizione 3.18. Un circuito si dice *euleriano* se attraversa tutti gli archi esattamente una volta. Un grafo si dice euleriano se ammette almeno un circuito euleriano.

Un esempio di un grafo euleriano è quello in figura.

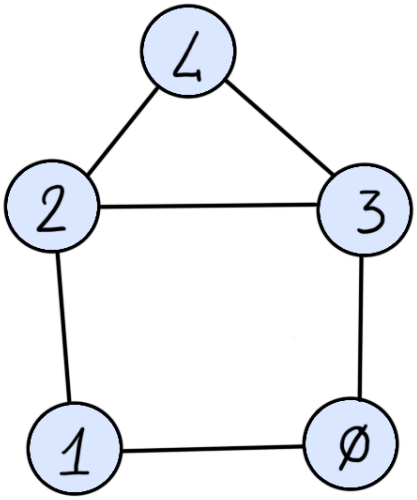


Figura 3.16: Esempio di grafo euleriano.

Vogliamo riuscire a risolvere il seguente problema.

Problema 3.19. Dato un grafo non orientato $\mathcal{G}$, stabilire se è euleriano.

Questo non è un problema di ottimizzazione, quindi è difficile definire un algoritmo ricorsivo o di programmazione dinamica per risolverlo. I problemi di questo tipo prendono il nome di *problemi di decisione*.

La condizione necessaria e sufficiente affinché un grafo sia euleriano è molto semplice.

Teorema 3.20. *Un grafo è euleriano se ogni nodo ha grado pari.*

Una giustificazione è molto semplice: ogni circuito euleriano che entra in un nodo deve avere un arco uscente sempre diverso.

Esempio 3.21. Il grafo in in figura non può essere euleriano

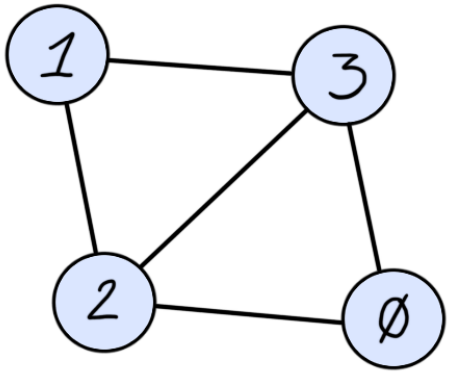


Figura 3.17: Esempio di grafo non euleriano.

perchè, un circuito euleriano deve entrare nel nodo 2 tramite tramite l'arco $\{1, 2\}$. Se questo cammino uscisse dal nodo 2 tramite l'arco $\{2, 3\}$, allora dovrebbe rientrare tramite l'arco $\{2, 3\}$, ma da qui non riuscirebbe più ad uscire e, dato che il cammino non è partito dal nodo 2, non è un circuito euleriano. Si può ripetere il ragionamento con ogni arco iniziale.

Con il teorema 3.20 possiamo definire un algoritmo per determinare se un grafo è euleriano.

Algoritmo 3.4.1: Algoritmo ricorsivo per risolvere il problema 3.19

```
1 Algoritmo: euleriano(G = (V, E))
2 foreach nodo v ∈ V do
3     if d(v) è dispari then
4         Il grafo non è euleriano;
5         return false;
6 Il grafo è euleriano;
7 return true;
```

Una volta appurato che esiste un circuito euleriano, è facile costruirlo. Basta prendere tutti gli archi del grafo.

Un ragionamento del tutto simile vale per i cammini euleriani.

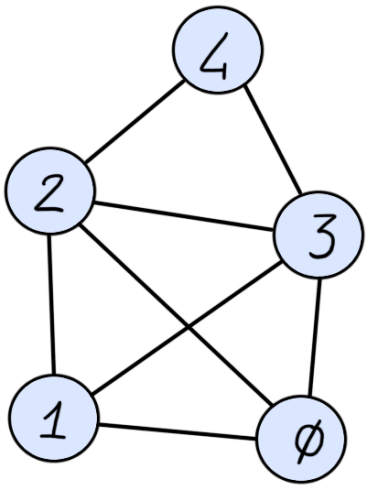


Figura 3.18: Esempio di cammino euleriano.

Dato che questa volta stiamo parlando di cammino e non di circuito il teorema 3.20 deve contenere una condizione meno rigida.

Teorema 3.22. *Un grafo $\mathcal{G}$ contiene un cammino euleriano se esistono esattamente due nodi di grado dispari.*

Un'altra famiglia importante di grafi è quella dei grafi Hamiltoniani.

Definizione 3.23. Un circuito si dice *hamiltoniamo* se visita ogni nodo esattamente una volta. Un grafo si dice hamiltoniano se ammette almeno un circuito hamiltoniano.

Immaginiamo di essere un postino che deve consegnare delle lettere. Un grafo non orientato può rappresentare la rete stradale e i pesi degli archi possono rappresentare la lunghezza del tragitto tra i due nodi. Il problema del *TSP* (Travelling Salesman Problem) consiste nel trovare, se esiste, un circuito che visiti tutti i nodi esattamente una volta e che il suo costo sia minimo.

Problema 3.24. In un grafo pesato non orientato $\mathcal{G} = (V, E, \psi)$, determinare un circuito hamiltoniano di costo minimo.

Questo problema è molto simile al problema 3.19, anche se la difficoltà è ben superiore.

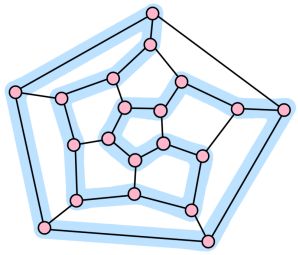


Figura 3.19: Esempio di grafo hamiltoniano.

Una condizione sufficiente, ma non necessaria all'esistenza di un tale circuito è garantita dal teorema di Ore.

Teorema 3.25. *Sia $\mathcal{G} = (V, E, \psi)$ un grafo con $n > 3$ nodi. Se, per ogni coppia di nodi non adiacenti $v, w \in V$ vale $d(v) + d(v) \geq n$, allora $\mathcal{G}$ è hamiltoniano.*

Questo teorema è stato generalizzato con il teorema di *Bondy-Chvátal*.

Individuare l'esistenza di un circuito hamiltoniano è relativamente semplice, ma la costruzione di un tale circuito è decisamente più complicata e non argomentod di questo corso.

Ci sono diversi algoritmi per risolvere il problema. Un primo approccio è quello greedy: Ordino gli archi per peso crescente e li inserisco nel cammino solo se non rispettano le condizioni di circuito hamiltoniano. Esistono anche diversi approcci euristici che forniscono una stima del peso del cammino ottimo.

3.5 Min-Path problem

Il prossimo problema che consideriamo è:

Problema 3.26 (Min-path problem)**.** Data una matrice $K \in \mathscr{M}(n \times m, \mathbb{N})$, determinare una successione di n celle, una per riga, tali che se le celle sono in due righe consecutive, si trovano o nella stessa colonna, o in colonne adiacenti.

L'algoritmo ricorsivo è semplice:

- per raggiungere una cella nella prima riga e nella colonna j, il cammino minimo ha costo $K_{1,j}$;
- per raggiungere la cella (i, j), allora il cammino minimo ha costo $K_{i,j}$ più il minimo dei costi per raggiungere le celle $(i-1, j-1), (i-1, j)$ e $(i-1, j+1)$ (con la convenzione che le celle fuori dalla matrice hanno costo $+\infty$).

L'algoritmo riceve in input una matrice di interi positivi K $n \times m$ e due interi $i < n$ e $j < m$ e manda in output il valore del cammino minimo che attraversa la matrice.

Algoritmo 3.5.1: Algoritmo ricorsivo per risolvere il problema 3.26

Algoritmo: $\mathrm{MP}(i, j)$
if $i = 1$ **then**
  return $K_{i,j}$;
if $j < 1 \vee j > n$ **then**
  return $+\infty$;
return $K_{i,j} + \min\{\mathrm{MP}(i-1, j-1), \mathrm{MP}(i-1, j), \mathrm{MP}(i-1, j+1)\}$;

Il valore ottimo del problema è ottenuto come

$$\min_{j=1,\dots,m} \{\mathrm{MP}(n, j)\}.$$

Ovviamente, come per gli algoritmi precedenti, molte chiamate ricorsive sono ridondanti, ma questo problema può essere risolto con la memoization, aggiungendo come input una struttura dati di appoggio. La matrice $M \in \mathscr{M}(n \times m, \mathbb{N})$, inizializzata a 0, contiene in posizione (i, j), il costo del cammino minimo per raggiungere la cella (i, j) in K.

```
int MPMem(int i, int j){
  if(T[i][j] == -1){
    if(i == 0) T[i][j] = K[i][j];
    else if(j == 0 || j == m+1) T[i][j] = INT_MAX;
    else T[i][j] = K[i][j] + min(MPMem(i-1,j-1),MPMem(i-1,j),
   MPMem(i-1,j+1));
  }
  return T[i][j];
}
```

Come nel caso ricorsivo, il valore ottimo del problema è ottenuto come

$$\min_{j=1,\dots,m} \{\texttt{MPMem(n-1,j)}\}.$$

Anche questo algoritmo può essere convertito in un algoritmo di programmazione dinamica. Modifichiamo la matrice K aggiungendo le colonne 0 e $m+1$ ponendole uguali a -1. La matrice M è $n \times m+2$, $K_1 \equiv M_1$, $K^0 \equiv M^0$ e $K^{m+1} \equiv M^{m+1}$. Il valore ottimo questa volta è il minimo di M_n.

```
int MPDin(){
  for(int i = 1 ; i < n ; i++)
    for (int j = 1; j < m+1; j++)
      M[i][j] = K[i][j] + min(M[i-1][j-1],M[i-1][j],M[i-1][j
   -1]);
```

```
5   int min = M[n-1][1];
6   for(int i = 2 ; i < m+1 ; i++)
7     if(min > M[n-1][i])
8       min = M[n-1][i];
9   return min;
10 }
```