*U*niversità degli *S*tudi di *U*rbino

Facolta' Di Scienze Matematiche Fisiche e Naturali
C. D. L. In informatica applicata
Algoritmi e strutture dati

Relazione progetto di laboratorio

Elvis Ciotti
Matr. 200244

11 giugno 2003

Prof. Marco Bernardo

Fasi dello sviluppo del software secondo la metodologia in the small

Fase 1: **Specifica del problema**

- Un labirinto è rappresentabile attraverso una matrice di lato n, tale che la casella di coordinate (i, j) è attraversabile se il valore ivi memorizzato è 1, non attraversabile se il valore ivi memorizzato è 0. Le caselle attraversabili devono essere almeno due, per consentire la designazione di una casella di partenza e di una casella di arrivo.
- Scrivere un programma ANSI C che, preso in input un intero n non inferiore a 3, genera casualmente un labirinto di dimensione n e lo riporta in output assieme a tutti i cammini semplici dalla casella di partenza alla casella di arrivo. Valutare per via sperimentale la complessità del programma.

***Fase 2 : A*nalisi del problema**

Input del problema:

L'input del problema consiste in un numero intero non inferiore a 3.
L'unico controllo da fare in questo caso è che il numero sia composto solo da cifre decimali e che il numero sia maggiore o uguale a 3.

Output del problema:

L'algoritmo deve generare una matrice di lato n in cui ogni casella di coordinate (i,j) vale casualmente 0 (casella non attraversabile) oppure 1 (casella attraversabile). Nel rispetto della specifica del problema le caselle attraversabili devono essere almeno 2 per consentire la designazione di una casella di partenza e una di arrivo, quindi alla fine della generazione bisogna scegliere due coppie di coordinate definite come arrivo e partenza da impostare a 1 indipendentemente dal valore che già abbiano. In questo modo si ha la certezza che ci siano almeno due caselle attraversabili.

Durante la generazione casuale della matrice, le possibilità che una casella sia attraversabile o che non lo sia sono uguali, poiché nella specifica del problema non è stato indicato.

Nella casella di coordinate (i, j), l'indice i è l'indice di riga, l'indice j è l'indice di colonna.
Nel caso n=3 la numerazione delle righe e delle colonne avviene partendo da 0 a n-1 (cioè 2).

↓ i	-- j ->		
	(0,0)	(0,1)	(0,2)
	(1,0)	(1,1)	(1,2)
	(2,0)	(2,1)	(2,2)

Il programma in oggetto deve calcolare tutti i cammini semplici dalla casella di partenza a quella di arrivo all'interno della matrice (considerata come un labirinto attraversabile nelle caselle il cui valore è 1).

Per percorsi semplici s'intendono delle successioni di coppie di coordinate
(i_0, j_0) -> (i_1, j_1) -> (i_2, j_2) -> (i_3, j_3) -> … -> (i_{n-1}, j_{n-1}) -> (i_n, j_n)
in cui :

- le coordinate (i_0, j_0) sono quelle di partenza e le coordinate (i_n, j_n) sono quelle di arrivo;
- tutte le caselle di coordinate (i_k, j_k) devono avere la coppia precedente (i_{k-1}, j_{k-1}) tale che $(i_k = i_{k-1}$ e $j_k = j_{k-1} \pm 1)$ oppure $(i_k = i_{k-1} \pm 1$ e $j_k = j_{k-1})$, ovvero i percorsi si devono muovere nelle quattro direzioni (che nel resto di questa relazione e nell'implementazione vengono identificate con le coordinate nord , est, sud, ovest);
- ogni coppia di coordinate deve essere presente una sola volta all'interno di un cammino semplice;
- Ogni coppia di coordinate (i_k, j_k) deve essere chiaramente compresa all'interno della matrice generata, cioè $0<k<n$.

Data una casella X di coordinate (i, j) la casella a nord di X ha coordinate (i-1, j), quella a est ha coordinate (i, j+1), quella a sud ha coordinate (i+1, j) e quella a ovest ha coordinate (i, j-1).

	NORD	
OVEST	X (i,j)	EST
	SUD	

Il caso in cui le coordinate della casella di partenza (i_k, j_k) coincidono con quelle di arrivo (i_k, j_k) è un percorso di lunghezza zero, nel rispetto della definizione di cammino semplice, costituito dalla sola coppia di coordinate (i_k, j_k).
La valutazione sperimentale della complessità del programma consiste nel misurare il tempo medio di esecuzione del programma al variare della dimensione del labirinto.

Fase 3 : Progettazione dell'algoritmo

I passi che il programma esegue per risolvere il problema sono i seguenti :

Acquisizione dell'input relativo alla dimensione del labirinto

L' input del problema è la dimensione del labirinto.
Il numero acquisito deve essere valido e maggiore o uguale a 3. La richiesta dell'input deve continuare finché non si introduce un input valido. Per numero valido si intende un numero composto da sole cifre decimali.

Dettaglio relativo all'implementazione: Per controllare l'input introdotto da linea di comando, l'acquisizione avviene in una stringa di 5 char (sufficienti per labirinti di lato < 10^6) che vengono convertiti in un intero dalla funzione atoi. Nel caso la stringa non contenga un numero valido (come per esempio una sequenza di caratteri che non sono cifre decimali), la funzione atoi restituisce falso e l'input viene chiesto nuovamente.

Generazione casuale del labirinto e delle caselle di partenza e arrivo

Dopo aver acquisito la dimensione del labirinto, viene generata una matrice in cui ogni casella viene casualmente impostata attraversabile (a '1') o non attraversabile (a '0').

Nell'implementazione in linguaggio C, il seme viene inizializzato utilizzando l'ora di sistema (libreria time.h) in modo da avere labirinti diversi ad ogni esecuzioen del programma.

Alla fine della generazione del labirinto vengono scelti casualmente le coordinate di arrivo e partenza all'interno del labirinto (cioè fra 0 e n-1).

Per semplificare il calcolo dei percorsi nel caso di caselle agli estremi del labirinto, tutta la generazione del labirinto e delle coordinate di arrivo e partenza avviene in realtà all'interno di una matrice di lato n+2, in modo da avere una cornice esterna di caselle non attraversabili. In fase di stampa verrà stampata solo la matrice interna e i percorsi si riferiranno alle coordinate relative, in modo del tutto trasparente all'utilizzatore. Questo metodo consente di semplificare i calcoli di generazione dei percorsi nel caso in cui ci si trovi in una casella agli estremi del labirinto; in tal caso occorrerebbe non proseguire all'infuori del labirinto e inserire un'ulteriore controllo in modo che i percorsi non escano dal labirinto. Il costo dell'allargamento della matrice è a favore del tempo di elaborazione e della semplificazione dell'algoritmo, a scapito solo di un leggero aumento lineare nell'uso di memoria [vengono occupate in più solo un numero di caselle pari a (n + 2) x 4 - 4 = 4n +4].

La scelta delle caselle di partenza e arrivo avviene nella matrice interna.
L'elaborazione e la stampa riguarderanno solo le caselle con indici da **1** a **n** (compresi).
L'elemento di coordinate (i, j) verrà riportato in output con coordinate (i-1, j-1) che si riferiscono alla matrice interna stampata.

0	0	0	0	0
0	**1/0**	**1/0**	**1/0**	0
0	**1/0**	**1/0**	**1/0**	0
0	**1/0**	**1/0**	**1/0**	0
0	0	0	0	0

Matrice con n = 3

Nell'implementazione questi primi due passi sono contenuti nella funzione `void crea_matrice(...)` che genera anche il labirinto e le caselle di partenza e arrivo.

Stampa del labirinto generato e delle coordinate di partenza e arrivo

Dopo la generazione del labirinto e della coordinate di partenza e arrivo, il tutto viene riportato in output. La stampa del labirinto riguarda solo la matrice interna escludendo la cornice. La coordinate di partenza e arrivo vengono stampate (così come i percorsi) con indici diminuiti di 1 in modo da avere le coordinate relative alla matrice interna.
Nella implementazione, la stampa è affidata alla funzione `void stampa_mat_e_perc(…)`.

Calcolo di tutti i percorsi semplici dalla casella di partenza a quella di arrivo e loro stampa

Per semplificare l'algoritmo, la ricerca dei percorsi viene effettuata a partire dalla casella di arrivo fino a cercare la casella di partenza. I percorsi vengono memorizzati in un albero in cui la radice contiene le coordinate della casella di arrivo e le foglie possono contenere le coordinate della casella di partenza. La costruzione dell'albero, così come la stampa dei percorsi e la rimozione dell'albero, avviene dinamicamente durante la ricerca dei percorsi senza che l'albero sia completamente presente in memoria.

Nel caso vengano trovati in un nodo la coordinate della casella di partenza, viene stampato il percorso salendo fino alla radice (arrivo) e stampando il contenuto delle coordinate dei nodi intermedi. Poi viene rimosso subito il ramo terminale che non serve più.
Nel caso le foglie (nodi terminali) non contengano invece le coordinate di partenza, il ramo terminale viene solo rimosso senza stampare nulla.

Questo è possibile poiché la funzione ricorsiva per ogni nodo esegue in dettaglio i seguenti passi:
- Scrive le coordinate, imposta il puntatore per il ritorno al padre, scrive che direzione ha preso il padre (per poter risalire e sapere da dove si era venuti).
- Se il nodo contiene le coordinate di partenza, stampa il percorso risalendo fino alla radice e rimuove* il ramo terminale che non serve più.
- Se il nodo non contiene le coordinate di partenza, controlla prima tutte le 4 direzioni e memorizza quali sono attraversabili**, poi se nessuna è attraversabile (nodo isolato), rimuove il nodo terminale. Se invece ci sono delle direzioni percorribili, viene richiamata la funzione di creazione del nodo sulle caselle adiacenti attraversabili, passandogli le coordinate adiacenti (che verrano ivi scritte), il puntatore corrente (che verrà usato come padre) e la direzione presa (che verrà scritta e servirà in caso di rimozione per salire e liberare il nodo da cui si è provenuti).
La funzione controlla all'inizio se la memoria è disponibile, altrimenti notifica e termina l'esecuzione.

* La rimozione avviene nel seguente modo: dato il puntatore, finchè il nodo ha tutti i figli non attraversabili, sale e libera il nodo; l'algoritmo di rimozione si ferma quando trova un nodo che ha almeno una direzione libera ancora da visitare e imposta la direzione ultima dal quale è provenuto come non attraversabile.
** Nel rispetto della definizione di percorso semplice (vedi fase 2: Analisi del Problema), un nodo di coordinate (i,j) associato alla casella di coordinate medesime, può rilanciare la funzione sulle caselle adiacenti nord (i-1, j), est (i, j+1), sud (i+1, j), ovest (i, j-1) se queste sono impostate attraversabili e se quelle coordinate non sono già presenti fra quel nodo e la radice (cioè nel percorso corrente). Quest'ultima condizione evita di calcolare i percorsi non semplici (cioè quelli che ripetono più di una volta la stessa coppia di coordinate).

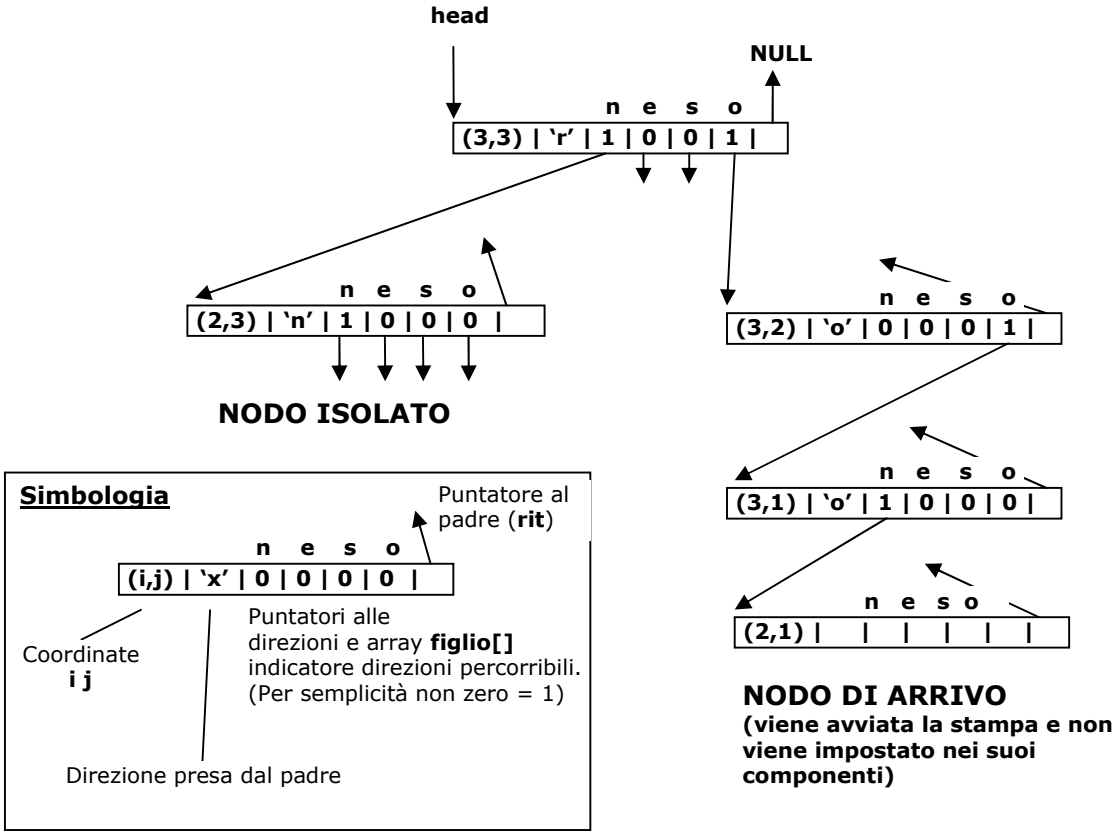
Con dettagli relativi alla implementazione, ogni nodo contiene quindi:
• Le coordinate della casella: *due variabili **int i** e **j**)*
• L'indicatore della direzione presa dal padre*: variabile **char** di nome **padre** che vale **'r'** se il nodo è la radice, **'n'** se il padre ha preso la direzione nord per arrivare al nodo corrente, **'e'** se il padre ha preso la direzione est per arrivare al nodo corrente e così via per sud **'s'** (sud) e ovest **'o'** (ovest)*
• L'indicatore di quali direzioni sono percorribili: *array di nome **figlio** di dimensione 4 è utilizzato in questo modo:*
  *se la direzione nord è percorribile, **figlio[0]** vale **non zero**, altrimenti **zero**;*
  *se la direzione est è percorribile, **figlio[1]** vale **non zero**, altrimenti **zero**;*
  *se la direzione sud è percorribile, **figlio[2]** vale **non zero**, altrimenti **zero**;*
  *se la direzione ovest è percorribile, **figlio[3]** vale **non zero**, altrimenti **zero**;*
Si utilizza per annullare la direzione di un nodo al quale si è saliti rimuovendo i figli a indicare che la direzione non è più percorribile.
• I 4 puntatori alle direzioni **nord**, **est**, **sud**, **ovest** : *così chiamati e sono associati all'indicatore di direzione percorribile*
• Un puntatore al ritorno : c*hiamato **rit** e serve per risalire in caso di stampa o rimozione*

Nel caso in cui le caselle di arrivo e partenza coincidono, l'albero sarà costituito dalla sola radice e la funzione ricorsiva di costruzione sarà eseguita una sola volta.

Esempio di albero con n=3 e un percorso trovato da (2,1) all'arrivo (3,3) con dettagli relativi all'implementazione
(In fase di stampa sarà stampata la matrice interna 3x3 e le coordinate sono diminuite di 1).

	0	1	2	3	4
0	0	0	0	0	0
1	0	**0**	**0**	**0**	0
2	0	**1** **part**	**0**	**1**	0
3	0	**1**	**1**	**1** **arr**	0
4	0	0	0	0	0

La costruzione dell'albero parte dalle coordinate di arrivo (3,3) e procede fermandosi nel ramo isolato (2,3) a sinistra e quando trova le coordinate di partenza (2,1) a destra.



La stampa relativa a questo percorso è:

```
Labirinto generato:
0  0  0
1  0  1
1  1  1
PERCORSI TROVATI DA (1,0) A (2,2):
[PERCORSO 1] : partenza -> 1,0 -> 2,0 -> 2,1 -> 2,2 -> arrivo.
```

(notare che le coordinate sono diminuite di uno e il labirinto è quello nel mezzo di dimensione 3 acquisita in input. Se si vogliono in stampa le coordinate in modo che gli indici riga e colonna partano da zero è sufficiente nella implementazione impostare il valore della costante `COORD_REL` a uno invece che a zero.

L'albero sopra rappresentato non risiede mai completamente in memoria.
Prima viene impostato tutto il nodo (3,3) impostando figlio[0] e figlio[3] a non zero, poi la funzione viene lanciata sulla direzione nord. Qui il nodo è isolato, infatti le caselle a nord, est e ovest non sono attraversabili e quella a sud è già presente fra la radice e il padre (in questo caso la radice è uguale al padre). L'array indicatore delle direzioni percorribili viene quindi impostato tutto a zero e viene rimosso il nodo terminale (2,3) con un puntatore che procede in alto fino a che non trova un nodo che non è isolato e imposta la direzione presa a zero (in questo caso il puntatore si ferma al padre sopra che ha la direzione ovest ancora da esplorare e la direzione nord viene impostata a zero poiché appena rimossa).
Adesso in memoria è presente solo il nodo (3,3) e viene lanciata la funzione sulle coordinate (3,2) , da qui l'unica direzione libera e ancora a ovest. Da infine la casella (3,1) si arriva da nord alla 2,1 che contiene le coordinate di partenza: viene quindi stampato il percorso e rimosso il ramo terminale, cioè tutto il resto dell'albero rimasto. L'algoritmo termina quindi restituendo l'unico percorso possibile.

Liberazione delle memoria utilizzata
Dopo la stampa di tutti i percorsi possibili. La memoria occupata dalla matrice viene rilasciata.
Nell'implementazione la liberazione e svolta dalla funzione `void disalloca_matrice(…)`.

Valutazione sperimentale della complessità del programma

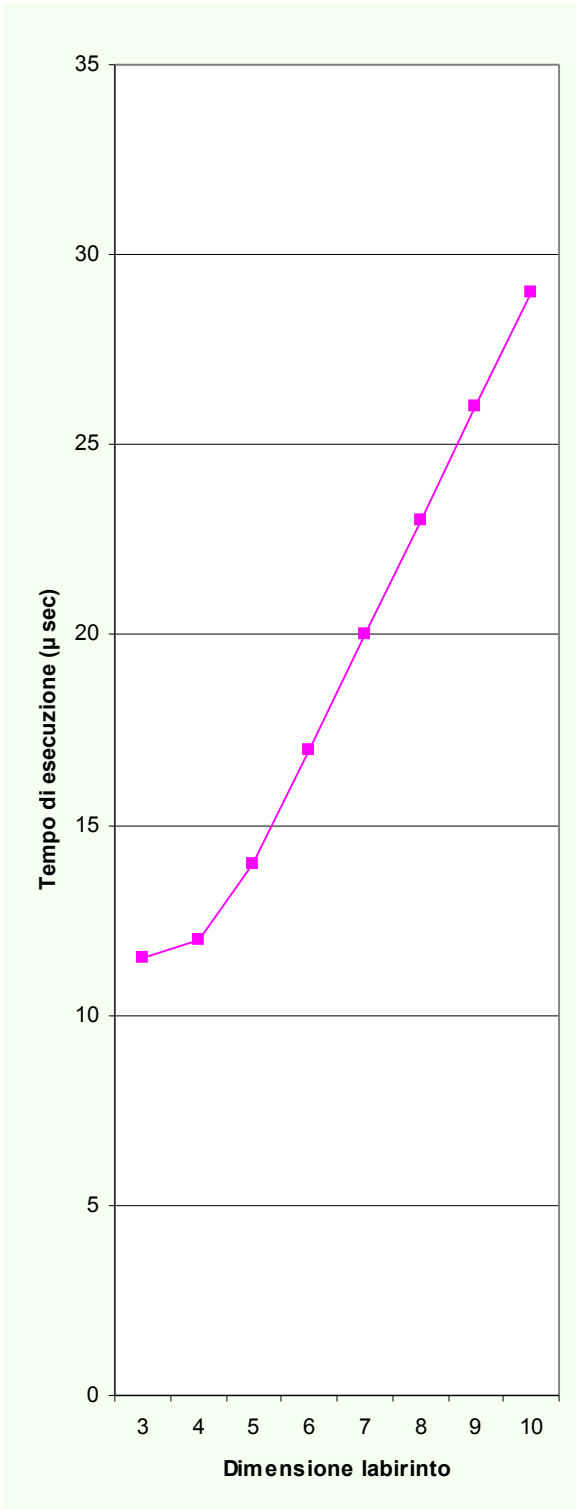
Come già accennato nella fase 2, si tratta di misurare il tempo medio di esecuzione in funzione della dimensione del labirinto.
Per ogni input n si è misurato il tempo medio dell'elaborazione per compiere 10^6 volte l'intero corpo del programma. Il tempo è quindi quello impiegato per generare la matrice, creare l'albero, risalire quando si sono trovati i percorsi, distruzione dei rami terminali e distruzione della matrice, il tutto 10^6 volte per ogni input n riportato nella prima colonna.
Il tempo medio si è dedotto da più misurazioni per ogni input della prima colonna, scartando quelli irrilevanti poiché troppo grandi o troppo piccoli rispetto alla media dei precedenti (il numero di misurazioni per ogni input è stato aumentato in proporzione al valore di n poiché al crescere di n si sono è rilevati sempre di più maggiori discordanze fra i valori nei tempi di elaborazione).
Il tempo nella seconda colonna è quindi riportato in milionesimi di secondo in riferimento a una sola esecuzione del corpo del programma.

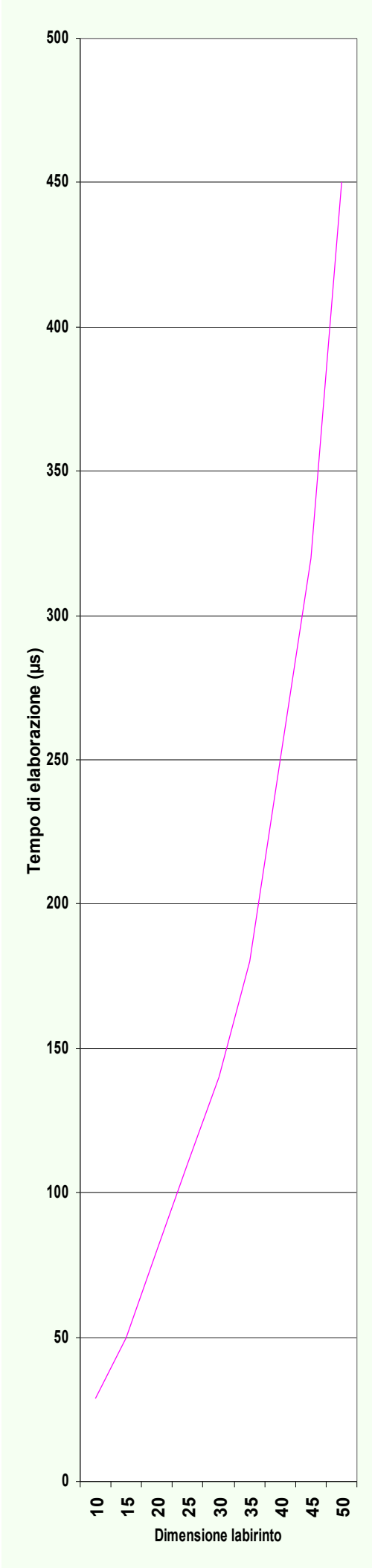
I dati raccolti in funzione di n sono i seguenti:

n	tempo (µs)	tempo/n
3	11,5	3,83
4	12	3,00
5	14	2,80
6	17	2,83
7	20	2,86
8	23	2,88
9	26	2,89
10	29	2,90
15	50	3,33
20	80	4,00
25	110	4,40
30	140	4,67
35	180	5,14
40	250	6,25
45	320	7,11
50	450	9,00

Sono riportati nel grafico sottostante i dati relativi a n che va da 3 a 10. Notare che per n inferiore a 10 il rapporto tempo/n (pendenza) è costante all'incirca a 3, a indicare una funzione di equazione y = 3x, quindi con complessità computazionale asintotica lineare O(n) finché l'input non supera 10.

Per n che va da 10 a 50 (grafico a destra) la complessità si può considerare polinomiale in n^2. La funzione è infatti compresa fra le curve $y=x^2/5$ e $y=x^2/7$ come si può verificare facilmente dalla tabella. (Il grafico della derivata prima che si approssima all'andamento di tempo/n in funzione della dimensione del labirinto, ha infatti un andamento lineare, all'incirca y = x/3 la cui primitiva è appunto $y= x^2/6$).
La complessità computazionale per n minore di 50 si può considerare quindi in $O(n^2)$.





Fase 4 : Implementazione dell'algoritmo in linguaggio ANSI C

Contenuto del file sorgente *prog.c*

```
/* Programma che, acquisito un numero intero n, genera casualmente un labirinto di lato n e trova tutti i cammini semplici dalla casella di partenza
a quella di arrivo (generate anch'esse casualmente). Il labirinto è una matrice in cui ogni casella è attraversabile nelle quattro direzioni se il
suo valore è 1.
  Studente: Elvis Ciotti   Prof. Marco Bernardo
  Algoritmi e Strutture Dati -  C.d.l. in Informatica Applicata - Facoltà SMFN - Urbino
  11 giugno 2003 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>


typedef struct albero {
   int i,j;
   char padre;  /* Direzione presa dal padre. 'n'=nord, 'e'=est, 's'=sud, 'o'=ovest, 'r'=radice */
   int figlio[4];  /* Indicatore esistenza figli se impostato a 1. [0]=nord, [1]=est, [2]=sud, [3]=ovest */
   struct albero *nord, *est, *sud, *ovest, *rit; /* Puntatori a i 4 figli e uno per il ritorno */
} albero_t;


#define COORD_REL 0 /* Se uguale a zero inizia il conteggio del numero di riga/colonna da 1, se impostato a 1 inizia da 0*/


void crea_matrice(int ***a, int *n, int *part_i, int *part_j, int *arr_i, int *arr_j);
void stampa_mat_e_perc(int **a, int n, int part_i, int part_j, int arr_i, int arr_j);
void crea_stamp_distr(int **a, int n, int corr_i, int corr_j, albero_t **p, albero_t **prec, int part_i, int part_j, int *n_perc, char
prov);
void stampa_per(albero_t *p, int *n_perc);
void rimuovi_term(albero_t **p);
int gia_pres(albero_t *p, int i, int j);
void disalloca_matrice(int ***a, int n);


int main(void)
{
   int **a = NULL,
       n,      /* Dimensione labirinto */
       part_i, part_j, arr_i, arr_j, /* Coordinate casella di partenza e arrivo */
       n_perc = 0; /* Contatore numero di percorsi trovati */
   albero_t *head = NULL,    /* Puntatore alla radice dell'albero*/
   *inizio = NULL;
```

```c
    /* Creazione matrice e coordinate di arrivo e partenza */
    crea_matrice(&a, &n, &part_i, &part_j, &arr_i, &arr_j);

    /* Stampa la matrice labirinto generata e le coordinate di arrivo e partenza */
    stampa_mat_e_perc(a, n, part_i, part_j, arr_i, arr_j);

    /* Inizia la costruzione-stampa-rimozione dinamica in modo ricorsivo dell'albero dei percorsi trovati */
    crea_stamp_distr(a, n, arr_i, arr_j, &head, &inizio, part_i, part_j, &n_perc, 'r');

    /* Disalloca la matrice labirinto */
    disalloca_matrice(&a, n+2);

    printf("Percorsi totali trovati: %d.\n", n_perc);

    return (0);
}


/* Acquisce la dimensione del labirinto e lo genera casualmente assieme alle caselle di arrivo e partenza */
void crea_matrice(int ***a, int *n, int *part_i, int *part_j, int *arr_i, int *arr_j)
{
    char temp[5]; /* Stringa temporanea per l'acquisizione del numero intero */
    int i, j;


    /* Acquisisco dimensione labirinto */
    do
    {
      printf("Inserisci dimensione matrice (maggiore o uguale a 3): ");
      scanf("%s", temp);
      *n = atoi(temp); /* Se la stringa non contiene un numero la funzione atoi restituisce 0 e viene richiesto di nuovo l'input */
    } while(*n < 3);

    /* Alloco memoria al puntatore passato per indirizzo per creare la matrice dinamica di dim n+2
      inizialmente azzerata */
    *a = (int **)calloc((size_t)*n+2, sizeof(int *));
    for (i=0; (i < *n+2); i++)
      (*a)[i] = (int *)calloc((size_t)*n+2, sizeof(int));

    /* Inizializzo il seme */
    srand((unsigned int)time(NULL));

    /* Genero labirinto casualmente (cornici escluse) */
    for (i=1; (i <= *n); i++)
      for (j=1; (j <= *n); j++)
         (*a)[i][j] = rand() % 2;
```

```c
    /* Genero coordinate di partenza scelte a caso nel labirinto */
    *part_i = rand() % *n + 1;
    *part_j = rand() % *n + 1;
    (*a)[*part_i][*part_j] = 1;
    /* Genero coordinate di arrivo scelte a caso nel labirinto */
    *arr_i = rand() % *n + 1;
    *arr_j = rand() % *n + 1;
    (*a)[*arr_i][*arr_j] = 1;
}


/* Stampa la matrice e le coordinate di arrivo e partenza */
void stampa_mat_e_perc(int **a, int n, int part_i, int part_j, int arr_i, int arr_j)
{
    int i,j;


    printf("Labirinto generato:\n");
    for (i = 1; (i <= n); i++)
    {
        for (j = 1; (j <= n); j++)
            printf("%3d", a[i][j]);
        printf("\n");
    }
    printf("\nPERCORSI TROVATI DA (%d:%d) A (%d:%d):\n", part_i-COORD_REL, part_j-COORD_REL, arr_i-COORD_REL, arr_j-COORD_REL);
}
```

```c
/* Creazione ricorsiva con stampa e distruzione dei percorsi memorizzati in una struttura ad albero */
void crea_stamp_distr(int **a, int n, int corr_i, int corr_j, albero_t **p, albero_t **prec, int part_i, int part_j, int *n_perc, char prov)
{
   /* Alloca memoria per il puntatore al nodo corrente*/
   *p = (albero_t *)malloc(sizeof(albero_t));
   /* Controllo disponibilità memoria */
   if (*p != NULL)
   {
      /* Scrivo coordinate, puntatore al padre e direzione presa dal padre in base ai parametri in input alla funzione */
      (*p)->i = corr_i;
      (*p)->j = corr_j;
      (*p)->rit = *prec;
      (*p)->padre = prov;
      /* Controllo caso in cui si è trovato la part in fondo all'albero con relativa stampa del perc e distr del ramo terminale */
      if ((corr_i == part_i) && (corr_j == part_j))
      {
         stampa_per(*p, &(*n_perc));
         rimuovi_term(&(*p));
      }
      else
      {
         /* Imposta il parametro figlio[n] che non è attraversabile se le coordinate nella direzione n sono a zero, oppure se le
            coordinate sono già presenti nel perc semplice, cioè fra il nodo (escluso) e la radice.(legge di De Morgan) */
         (*p)->figlio[0] = ((a[corr_i-1][corr_j] != 0) && !gia_pres(*p, corr_i-1, corr_j));
         (*p)->figlio[1] = ((a[corr_i][corr_j+1] != 0) && !gia_pres(*p, corr_i, corr_j+1));
         (*p)->figlio[2] = ((a[corr_i+1][corr_j] != 0) && !gia_pres(*p, corr_i+1, corr_j));
         (*p)->figlio[3] = ((a[corr_i][corr_j-1] != 0) && !gia_pres(*p, corr_i, corr_j-1));

         /* Eventuale rimozione del ramo terminale che non ha figli (e che non contiene la partenza in fondo) */
         if ((*p)->figlio[0] == 0 && (*p)->figlio[1] == 0 && (*p)->figlio[2] == 0 && (*p)->figlio[3] == 0)
           rimuovi_term(&(*p));
        else
         {
          /* Richiama questa funzione ricorsivamente nelle direzioni libere non ancora visitate */
            if ((*p)->figlio[0] != 0)
               crea_stamp_distr(a, n, corr_i-1, corr_j, &((*p)->nord), &(*p), part_i, part_j, &(*n_perc), 'n');
            if ((*p)->figlio[1] != 0)
               crea_stamp_distr(a, n, corr_i, corr_j+1, &((*p)->est), &(*p), part_i, part_j, &(*n_perc), 'e');
            if ((*p)->figlio[2] != 0)
               crea_stamp_distr(a, n, corr_i+1, corr_j, &((*p)->sud), &(*p), part_i, part_j, &(*n_perc), 's');
            if ((*p)->figlio[3] != 0)
               crea_stamp_distr(a, n, corr_i, corr_j-1, &((*p)->ovest), &(*p), part_i, part_j, &(*n_perc), 'o');
         }
      }
   }
   else
      printf("Memoria esaurita. Impossibile continuare.\n");
}
```

```c
/* Stampa il percorso a ritroso a puntatore al nodo passato per copia che contiene le coordinate di partenza */
void stampa_per(albero_t *p, int *n_perc)
{
   (*n_perc)++;


   printf("[ PERCORSO %d ] : partenza -> %d,%d ->", *n_perc, p->i-COORD_REL, p->j-COORD_REL);
   /* Scorrimento fino alla radice dell'albero  e stampa delle coordinate trovate */
   while ( p->padre != 'r')
   {
      p = p->rit;
      printf(" %d,%d ->", p->i-COORD_REL, p->j-COORD_REL);
   }
   printf(" arrivo.\n");
}


/* Rimuove ramo terminale puntato da 'p' */
void rimuovi_term(albero_t **p)
{
   int prov;
   albero_t *temp; /* Puntatore temporaneo per rimozione */


   /* Controlla il caso in cui il puntatore passato sia la testa dell'albero e in tal caso lo libera (avviene solo se le caselle di arrivo e
      partenza coincidono) */
   if ((*p)->padre == 'r')
   free(*p);
   else  /* se il puntatore non punta alla radice  */
   {
      /* Rimuove ramo terminale salendo finchè trova rami liberi o è alla radice. (De Morgan) */
      while( (*p)->figlio[0] == 0 && (*p)->figlio[1] == 0 && (*p)->figlio[2] == 0 && (*p)->figlio[3] == 0 && (*p)->padre != 'r' )
      {
         temp = *p; /* Copia il puntatore corrente */
         prov = temp->padre; /* Memorizza la direzione presa dal puntatore */
         (*p) = (*p)->rit; /* Salita di un livello del puntatore */
         free(temp); /* Free del puntatore copiato all'inizio*/

          /* Annullamento della direzione da cui proveniva il puntatore (prima di salire) */
         if (prov == 'n')
           (*p)->figlio[0] = 0;
         if (prov == 'e')
           (*p)->figlio[1] = 0;
         if (prov == 's')
           (*p)->figlio[2] = 0;
         if (prov == 'o')
           (*p)->figlio[3] = 0;
      }
   }
}
```

```c
/* Ricerca all'indietro delle coordinate (i,j) a partire dal nodo puntato (escluso); restituisce vero (non zero) se la coordinata è già presente */
int gia_pres(albero_t *p, int i, int j)
{
   int ris = 0;


   /* La ricerca prosegue finchè non si è alla radice o finchè non si è già trovata una coordinata uguale */
   while ( (p->padre != 'r') && !ris)
   {
      p = p->rit;
      ris = ( (p->i == i) && (p->j == j) );
   }
   return ris; /* Se non si sono trovate la coordinate, restituisce zero */
}


/* Disalloca un array bidimensionale dinamico di lato n, prendendo per indirizzo il puntatore originario */
void disalloca_matrice(int ***a, int n)
{
   int i;


   /* Disalloco memoria ai puntatori del vettore */
   for (i = 0; (i < n); i++)
      free((*a)[i]);
   /* Disalloco memoria al puntatore originario */
   free(*a);
}
```

Contenuto del file sorgente *makefile*

```
prog: prog.c
        gcc -ansi -Wall -O prog.c -o prog
```

Fase 5 : Testing del programma

Nel caso in cui introduciamo come input un numero non valido (cioè non composto da sole cifre decimali) o minore di 3, il valore da introdurre viene chiesto nuovamente.

Alcuni test effettuati:
per maggiore chiarezza, in esecuzione la costante COORD_REL è stata impostata a 1, i numeri di riga e colonna si intendono quindi partire da 1.

---

n=3

```
Labirinto generato:
  0  1  1
  1  1  1
  1  1  0
PERCORSI TROVATI DA (2:1) A (2:3):
[ PERCORSO 1 ] : partenza -> 2,1 -> 3,1 -> 3,2 -> 2,2 -> 1,2 -> 1,3 -> 2,3 -> arrivo.
[ PERCORSO 2 ] : partenza -> 2,1 -> 2,2 -> 1,2 -> 1,3 -> 2,3 -> arrivo.
[ PERCORSO 3 ] : partenza -> 2,1 -> 3,1 -> 3,2 -> 2,2 -> 2,3 -> arrivo.
[ PERCORSO 4 ] : partenza -> 2,1 -> 2,2 -> 2,3 -> arrivo.
Percorsi totali trovati: 4.

Labirinto generato:
  1  0  0
  1  1  0
  1  0  1
PERCORSI TROVATI DA (3:1) A (2:2):
[ PERCORSO 1 ] : partenza -> 3,1 -> 2,1 -> 2,2 -> arrivo.
Percorsi totali trovati: 1.

Labirinto generato:
  1  1  1
  0  1  1
  0  1  0
PERCORSI TROVATI DA (1:1) A (1:2):
[ PERCORSO 1 ] : partenza -> 1,1 -> 1,2 -> arrivo.
Percorsi totali trovati: 1.

Labirinto generato:
  0  0  1
  1  0  1
  0  1  1
PERCORSI TROVATI DA (2:1) A (2:3):
Percorsi totali trovati: 0.

Labirinto generato:
  0  0  1
  0  0  0
  1  1  0
PERCORSI TROVATI DA (3:1) A (3:1):          [CASO CON  PARTENZA =  ARRIVO]
[ PERCORSO 1 ] : partenza -> 3,1 -> arrivo.
Percorsi totali trovati: 1.
```

---

n=4

```
Labirinto generato:
  1  0  0  1
  0  1  1  1
  0  1  1  1
  0  0  1  1
PERCORSI TROVATI DA (1:1) A (2:4):
Percorsi totali trovati: 0.
```

```
Labirinto generato:
  0  0  0  0
  1  1  1  0
  1  1  0  1
  1  0  0  1
PERCORSI TROVATI DA (3:1) A (2:2):
[ PERCORSO 1 ] : partenza -> 3,1 -> 3,2 -> 2,2 -> arrivo.
[ PERCORSO 2 ] : partenza -> 3,1 -> 2,1 -> 2,2 -> arrivo.
Percorsi totali trovati: 2.

Labirinto generato:
  1  1  1  1
  1  0  0  0
  1  0  0  0
  1  1  1  0
PERCORSI TROVATI DA (4:3) A (4:1):
[ PERCORSO 1 ] : partenza -> 4,3 -> 4,2 -> 4,1 -> arrivo.
Percorsi totali trovati: 1.

Labirinto generato:
  1  1  1  1
  1  1  0  1
  1  0  1  1
  0  0  0  1
PERCORSI TROVATI DA (2:2) A (4:4):
[ PERCORSO 1 ] : partenza -> 2,2 -> 1,2 -> 1,3 -> 1,4 -> 2,4 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 2 ] : partenza -> 2,2 -> 2,1 -> 1,1 -> 1,2 -> 1,3 -> 1,4 -> 2,4 -> 3,4 -> 4,4
-> arrivo.
Percorsi totali trovati: 2.
```

---

n=5

```
Labirinto generato:
  0  0  1  1  0
  1  0  1  1  0
  1  1  1  0  0
  0  1  1  1  1
  0  0  0  1  1
PERCORSI TROVATI DA (3:1) A (1:3):
[ PERCORSO 1 ] : partenza -> 3,1 -> 3,2 -> 4,2 -> 4,3 -> 3,3 -> 2,3 -> 2,4 -> 1,4 -> 1,3
-> arrivo.
[ PERCORSO 2 ] : partenza -> 3,1 -> 3,2 -> 3,3 -> 2,3 -> 2,4 -> 1,4 -> 1,3 -> arrivo.
[ PERCORSO 3 ] : partenza -> 3,1 -> 3,2 -> 4,2 -> 4,3 -> 3,3 -> 2,3 -> 1,3 -> arrivo.
[ PERCORSO 4 ] : partenza -> 3,1 -> 3,2 -> 3,3 -> 2,3 -> 1,3 -> arrivo.
Percorsi totali trovati: 4.

Labirinto generato:
  1  0  1  0  1
  0  1  0  0  1
  1  1  1  1  0
  0  0  1  1  1
  1  1  1  0  1
PERCORSI TROVATI DA (3:4) A (5:3):
[ PERCORSO 1 ] : partenza -> 3,4 -> 3,3 -> 4,3 -> 5,3 -> arrivo.
[ PERCORSO 2 ] : partenza -> 3,4 -> 4,4 -> 4,3 -> 5,3 -> arrivo.
Percorsi totali trovati: 2.

Labirinto generato:
  1  1  0  1  0
  0  1  1  1  0
  0  1  1  1  1
  1  1  1  1  0
  0  0  0  1  1
PERCORSI TROVATI DA (3:3) A (4:4):
[ PERCORSO 1 ] : partenza -> 3,3 -> 2,3 -> 2,4 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 2 ] : partenza -> 3,3 -> 3,2 -> 2,2 -> 2,3 -> 2,4 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 3 ] : partenza -> 3,3 -> 4,3 -> 4,2 -> 3,2 -> 2,2 -> 2,3 -> 2,4 -> 3,4 -> 4,4
-> arrivo.
[ PERCORSO 4 ] : partenza -> 3,3 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 5 ] : partenza -> 3,3 -> 4,3 -> 4,4 -> arrivo.
```

```
[ PERCORSO 6 ] : partenza -> 3,3 -> 3,4 -> 2,4 -> 2,3 -> 2,2 -> 3,2 -> 4,2 -> 4,3 -> 4,4
-> arrivo.
[ PERCORSO 7 ] : partenza -> 3,3 -> 2,3 -> 2,2 -> 3,2 -> 4,2 -> 4,3 -> 4,4 -> arrivo.
[ PERCORSO 8 ] : partenza -> 3,3 -> 3,2 -> 4,2 -> 4,3 -> 4,4 -> arrivo.
Percorsi totali trovati: 8.
```

---

n=6

```
Labirinto generato:
  1  1  1  0  1  1
  1  1  1  1  1  1
  0  1  0  1  0  1
  1  1  1  1  0  1
  1  0  1  0  0  1
  0  1  0  1  1  0
PERCORSI TROVATI DA (5:6) A (4:4):
[ PERCORSO 1 ] : partenza -> 5,6 -> 4,6 -> 3,6 -> 2,6 -> 1,6 -> 1,5 -> 2,5 -> 2,4 -> 3,4
-> 4,4 -> arrivo.
[ PERCORSO 2 ] : partenza -> 5,6 -> 4,6 -> 3,6 -> 2,6 -> 2,5 -> 2,4 -> 3,4 -> 4,4 ->
arrivo.
[ PERCORSO 3 ] : partenza -> 5,6 -> 4,6 -> 3,6 -> 2,6 -> 1,6 -> 1,5 -> 2,5 -> 2,4 -> 2,3
-> 1,3 -> 1,2 -> 2,2 -> 3,2 -> 4,2 -> 4,3 -> 4,4 -> arrivo.
[ PERCORSO 4 ] : partenza -> 5,6 -> 4,6 -> 3,6 -> 2,6 -> 2,5 -> 2,4 -> 2,3 -> 1,3 -> 1,2
-> 2,2 -> 3,2 -> 4,2 -> 4,3 -> 4,4 -> arrivo.
[ PERCORSO 5 ] : partenza -> 5,6 -> 4,6 -> 3,6 -> 2,6 -> 1,6 -> 1,5 -> 2,5 -> 2,4 -> 2,3
-> 2,2 -> 3,2 -> 4,2 -> 4,3 -> 4,4 -> arrivo.
[ PERCORSO 6 ] : partenza -> 5,6 -> 4,6 -> 3,6 -> 2,6 -> 2,5 -> 2,4 -> 2,3 -> 2,2 -> 3,2
-> 4,2 -> 4,3 -> 4,4 -> arrivo.
[ PERCORSO 7 ] : partenza -> 5,6 -> 4,6 -> 3,6 -> 2,6 -> 1,6 -> 1,5 -> 2,5 -> 2,4 -> 2,3
-> 1,3 -> 1,2 -> 1,1 -> 2,1 -> 2,2 -> 3,2 -> 4,2 -> 4,3 -> 4,4 -> arrivo.
[ PERCORSO 8 ] : partenza -> 5,6 -> 4,6 -> 3,6 -> 2,6 -> 2,5 -> 2,4 -> 2,3 -> 1,3 -> 1,2
-> 1,1 -> 2,1 -> 2,2 -> 3,2 -> 4,2 -> 4,3 -> 4,4 -> arrivo.
Percorsi totali trovati: 8.
```

---

n=8

```
Labirinto generato:
  1  0  0  1  1  1  0  1
  1  0  1  1  0  0  0  0
  1  1  1  1  0  0  0  0
  0  1  0  1  1  1  1  0
  1  1  1  0  0  0  0  1
  0  1  1  1  1  1  1  1
  1  1  0  1  1  0  0  1
  1  0  0  1  1  1  1  1
PERCORSI TROVATI DA (8:8) A (4:4):
[ PERCORSO 1 ] : partenza -> 8,8 -> 7,8 -> 6,8 -> 6,7 -> 6,6 -> 6,5 -> 6,4 -> 6,3 -> 5,3
-> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 2,3 -> 2,4 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 2 ] : partenza -> 8,8 -> 8,7 -> 8,6 -> 8,5 -> 7,5 -> 6,5 -> 6,4 -> 6,3 -> 5,3
-> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 2,3 -> 2,4 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 3 ] : partenza -> 8,8 -> 8,7 -> 8,6 -> 8,5 -> 8,4 -> 7,4 -> 7,5 -> 6,5 -> 6,4
-> 6,3 -> 5,3 -> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 2,3 -> 2,4 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 4 ] : partenza -> 8,8 -> 7,8 -> 6,8 -> 6,7 -> 6,6 -> 6,5 -> 7,5 -> 7,4 -> 6,4
-> 6,3 -> 5,3 -> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 2,3 -> 2,4 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 5 ] : partenza -> 8,8 -> 8,7 -> 8,6 -> 8,5 -> 7,5 -> 7,4 -> 6,4 -> 6,3 -> 5,3
-> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 2,3 -> 2,4 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 6 ] : partenza -> 8,8 -> 7,8 -> 6,8 -> 6,7 -> 6,6 -> 6,5 -> 7,5 -> 8,5 -> 8,4
-> 7,4 -> 6,4 -> 6,3 -> 5,3 -> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 2,3 -> 2,4 -> 3,4 -> 4,4 ->
arrivo.
[ PERCORSO 7 ] : partenza -> 8,8 -> 8,7 -> 8,6 -> 8,5 -> 8,4 -> 7,4 -> 6,4 -> 6,3 -> 5,3
-> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 2,3 -> 2,4 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 8 ] : partenza -> 8,8 -> 7,8 -> 6,8 -> 6,7 -> 6,6 -> 6,5 -> 6,4 -> 6,3 -> 6,2
-> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 2,3 -> 2,4 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 9 ] : partenza -> 8,8 -> 8,7 -> 8,6 -> 8,5 -> 7,5 -> 6,5 -> 6,4 -> 6,3 -> 6,2
-> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 2,3 -> 2,4 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 10 ] : partenza -> 8,8 -> 8,7 -> 8,6 -> 8,5 -> 8,4 -> 7,4 -> 7,5 -> 6,5 -> 6,4
-> 6,3 -> 6,2 -> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 2,3 -> 2,4 -> 3,4 -> 4,4 -> arrivo.
```

```
[ PERCORSO 11 ] : partenza -> 8,8 -> 7,8 -> 6,8 -> 6,7 -> 6,6 -> 6,5 -> 7,5 -> 7,4 -> 6,4
-> 6,3 -> 6,2 -> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 2,3 -> 2,4 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 12 ] : partenza -> 8,8 -> 8,7 -> 8,6 -> 8,5 -> 7,5 -> 7,4 -> 6,4 -> 6,3 -> 6,2
-> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 2,3 -> 2,4 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 13 ] : partenza -> 8,8 -> 7,8 -> 6,8 -> 6,7 -> 6,6 -> 6,5 -> 7,5 -> 8,5 -> 8,4
-> 7,4 -> 6,4 -> 6,3 -> 6,2 -> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 2,3 -> 2,4 -> 3,4 -> 4,4 ->
arrivo.
[ PERCORSO 14 ] : partenza -> 8,8 -> 8,7 -> 8,6 -> 8,5 -> 8,4 -> 7,4 -> 6,4 -> 6,3 -> 6,2
-> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 2,3 -> 2,4 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 15 ] : partenza -> 8,8 -> 7,8 -> 6,8 -> 6,7 -> 6,6 -> 6,5 -> 6,4 -> 6,3 -> 5,3
-> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 16 ] : partenza -> 8,8 -> 8,7 -> 8,6 -> 8,5 -> 7,5 -> 6,5 -> 6,4 -> 6,3 -> 5,3
-> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 17 ] : partenza -> 8,8 -> 8,7 -> 8,6 -> 8,5 -> 8,4 -> 7,4 -> 7,5 -> 6,5 -> 6,4
-> 6,3 -> 5,3 -> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 18 ] : partenza -> 8,8 -> 7,8 -> 6,8 -> 6,7 -> 6,6 -> 6,5 -> 7,5 -> 7,4 -> 6,4
-> 6,3 -> 5,3 -> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 19 ] : partenza -> 8,8 -> 8,7 -> 8,6 -> 8,5 -> 7,5 -> 7,4 -> 6,4 -> 6,3 -> 5,3
-> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 20 ] : partenza -> 8,8 -> 7,8 -> 6,8 -> 6,7 -> 6,6 -> 6,5 -> 7,5 -> 8,5 -> 8,4
-> 7,4 -> 6,4 -> 6,3 -> 5,3 -> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 21 ] : partenza -> 8,8 -> 8,7 -> 8,6 -> 8,5 -> 8,4 -> 7,4 -> 6,4 -> 6,3 -> 5,3
-> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 22 ] : partenza -> 8,8 -> 7,8 -> 6,8 -> 6,7 -> 6,6 -> 6,5 -> 6,4 -> 6,3 -> 6,2
-> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 23 ] : partenza -> 8,8 -> 8,7 -> 8,6 -> 8,5 -> 7,5 -> 6,5 -> 6,4 -> 6,3 -> 6,2
-> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 24 ] : partenza -> 8,8 -> 8,7 -> 8,6 -> 8,5 -> 8,4 -> 7,4 -> 7,5 -> 6,5 -> 6,4
-> 6,3 -> 6,2 -> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 25 ] : partenza -> 8,8 -> 7,8 -> 6,8 -> 6,7 -> 6,6 -> 6,5 -> 7,5 -> 7,4 -> 6,4
-> 6,3 -> 6,2 -> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 26 ] : partenza -> 8,8 -> 8,7 -> 8,6 -> 8,5 -> 7,5 -> 7,4 -> 6,4 -> 6,3 -> 6,2
-> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 27 ] : partenza -> 8,8 -> 7,8 -> 6,8 -> 6,7 -> 6,6 -> 6,5 -> 7,5 -> 8,5 -> 8,4
-> 7,4 -> 6,4 -> 6,3 -> 6,2 -> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 3,4 -> 4,4 -> arrivo.
[ PERCORSO 28 ] : partenza -> 8,8 -> 8,7 -> 8,6 -> 8,5 -> 8,4 -> 7,4 -> 6,4 -> 6,3 -> 6,2
-> 5,2 -> 4,2 -> 3,2 -> 3,3 -> 3,4 -> 4,4 -> arrivo.
Percorsi totali trovati: 28.
```

Tutti i test sono risultati validi.
Una ulteriore verifica di correttezza dell'algoritmo è stata eseguita nella fase di calcolo sperimentale della complessità computazionale asintotica. Il corpo del programma è stato eseguito milioni di volte per labirinti di lato ,4,5,…,10,15,20,25,…,50 senza mai bloccarsi e rilasciando alla fine la memoria allo stato iniziale (controllo eseguito con monitoraggio delle risorse durante, prima e dopo l'esecuzione).