Francesco, Valentina, Laura Sisini e Annalisa Pazzi

Applicazioni di grafi e algoritmi alla fuga di Pac-Man dal Ghosts Team

Edizioni: i Sisini Pazzi, 2020

AI NOSTRI LETTORI

Scuola Sisini produce testi e giochi creati con l'obiettivo di divulgare argomenti complessi.

C'è un divario tra la richiesta e l'offerta di conoscenza: da un lato articoli scientifici destinati solo a un pubblico iper-specializzato, dall'altro la diffusione nozionale di contenuti affascinanti, che sono, tuttavia, descritti solo qualitativamente

Tra questi due estremi c'è il metodo di diffusione della Scuola Sisini che, attraverso un percorso ragionato, porta all'uscita dalla zona di comfort per creare strumenti e basi, che poi permettono di approfondire autonomamente gli argomenti di interesse.

Informazioni sulla proprietà intellettuale e licenza d'uso

/licenses/.

PIATTAFORMA SUPPORTATA

Il codice presentato nel libro è standard C (C99) e può essere compilato su ciascuna piattaforma. Comunque, l'abbiamo testato solo su Linux usando il compilatore GCC, e questa è la configurazione raccomandata per compilare ed eseguire il codice. Il gioco Pac-Man qui presentato si basa sulla grafica del terminale ma, per Windows 10, sembra si debba abilitare il terminale virtuale perché venga mostrata correttamente la griglia di gioco.

Per chi è interessato all'utilizzo di Windows, si consiglia di consultare la documentazione Microsoft riguardo a: "Abilitazione dell'elaborazione del terminale virtuale"

INDICE

INTRODUZIONE

"Pac-Man può essere considerato il capostipite dell'età d'oro dei videogiochi arcade", come sottolineato da Philipp Rohlfshagen (p1). Il suo impatto sul mondo commerciale dei videogiochi è stato immediatamente accompagnato da un interesse di natura scientifica e, negli ultimi vent'anni, il gioco è stato l'oggetto principale di numerose ricerche e studi.
Pac-Man è adatto per essere un banco di prova efficiente nella ricerca scientifica, in particolare nel campo dell'intelligenza artificiale, della robotica, dello sviluppo di interfacce cervello-computer, della biologia, di psicologia e sociologia. È un dato di fatto che Pac-Man ha fornito una piattaforma per sviluppare e testare le tecniche e le tecnologie di oggi.

L'idea di questo libro è di usare il gioco Pac-Man per indagare e analizzare il campo emergente della teoria dei grafi. Sebbene il libro sia destinato ai principianti, riteniamo che sia interessante anche per coloro che già conoscono la teoria dei grafi. La lettura richiede una buona conoscenza del linguaggio di programmazione C e non sono state utilizzate altri linguaggi di programmazione.

Nel testo vengono introdotti i seguenti concetti:

- Grafi, vertici ed archi
- Grado del grafo
- Isomorfismi tra grafi
- Percorsi, attraversamenti e coperture di grafi
- Cammino euleriano
- Scomposizione cellulare di un campo
- Algoritmo di Boustrophedon
- Algoritmo A*

- Algoritmo Breadth first Search

SORGENTI, TIPI E FUNZIONI

Tutti i codici completi si trovano in fondo al testo nelle appendici e sono anche disponibili su GitHub nel repository specificato in fondo al libro.

La piattaforma per il gioco è codificata usando nomi per variabili, tipi e funzioni dalla lingua italiana. Ad ogni modo, visto che è stata prodotta anche una versione in inglese del testo, il codice è commentato completamente anche in inglese.

Per le funzioni e i tipi di dato, utilizzati per creare, manipolare e analizzare grafici, si è creata una libreria ad hoc i cui nomi sono in latino classico.
Il latino è una lingua *declinata* e questo consente di esprimere una frase significativa utilizzando solo poche parole che possono essere utilizzate per creare nomi per tipi e funzioni.
Le basi del latino per comprendere il significato di tipi e funzioni sono presentate nell'Appendice di questo libro.

LA PIATTAFORMA DI GIOCO

Pac-Man è un noto videogioco arcade. Per coloro che non hanno mai giocato a Pac-Man, diciamo che è un videogioco di tipo arcade a labirinto. È stato distribuito dalla Namco nel 1980. Pac-Man, il protagonista, deve mangiare le pillole disposte lungo il percorso del labirinto mentre è inseguito da quattro fantasmi (il ghost team) che cercano di bloccarlo.

In questo capitolo presentiamo un'implementazione del gioco Pac-Man scritto in linguaggio C allo scopo di presentare alcune applicazioni della teoria dei grafi.
Questa implementazione è progettata per essere molto simile all'originale ma, ad ogni modo, ci sono diverse differenze importanti:

- La velocità di Pac-Man e dei fantasmi è identica
- Pac-Man non rallenta quando mangia le pillole
- Nel labirinto non ci sono frutti
- Le pillole energetiche hanno l'unico effetto per impedire ai fantasmi di mangiare Pac-Man, ma Pac-Man non può mangiare i fantasmi.

Il labirinto

Il labirinto è costituito da una griglia di celle di 31 linee per 28 colonne. Ogni cella può essere:

- occupata da un muro
- una pillola
- una pillola energetica
- o essere vuota.

Il labirinto è strutturato in modo da essere attraversato da

corridoi che non sono mai più larghi di una singola cella.
Il labirinto è rappresentato come una matrice di tipo `oggetto`.

```
  oggetto  campo[ALTEZZA][LARGHEZZA]={
  {J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J},
  {J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J},
  {A,E,E,E,E,E,E,E,E,E,E,E,E,I,L,E,E,E,E,E,E,E,E,E,E,E,E,E,B},
  {F,U,U,U,U,U,U,U,U,U,U,U,U,S,S,U,U,U,U,U,U,U,U,U,U,U,U,U,X},
  {F,U,G,T,T,H,U,G,T,T,T,H,U,S,S,U,G,T,T,T,H,U,G,T,T,H,U,X},
  {F,V,S,J,J,S,U,S,J,J,J,S,U,S,S,U,S,J,J,J,S,U,S,J,J,S,V,X},
  {F,U,W,T,T,Y,U,W,T,T,T,Y,U,W,Y,U,W,T,T,T,Y,U,W,T,T,Y,U,X},
  {F,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,X},
  {F,U,G,T,T,H,U,G,H,U,G,T,T,T,T,T,T,H,U,G,H,U,G,T,T,H,U,X},
  {F,U,W,T,T,Y,U,S,S,U,W,T,T,H,G,T,T,Y,U,S,S,U,W,T,T,Y,U,X},
  {F,U,U,U,U,U,U,S,S,U,U,U,U,S,S,U,U,U,U,S,S,U,U,U,U,U,U,X},
  {C,Z,Z,Z,Z,B,U,S,W,T,T,H,J,S,S,J,G,T,T,Y,S,U,A,Z,Z,Z,Z,D},
  {J,J,J,J,J,F,U,S,G,T,T,Y,J,W,Y,J,W,T,T,H,S,U,X,J,J,J,J,J},
  {J,J,J,J,J,F,U,S,S,J,J,J,J,J,J,J,J,J,J,S,S,U,X,J,J,J,J,J},
  {J,J,J,J,J,F,U,S,S,J,A,E,E,J,J,E,E,B,J,S,S,U,X,J,J,J,J,J},
  {Z,Z,Z,Z,Z,D,U,W,Y,J,F,J,J,J,J,J,J,X,J,W,Y,U,C,E,E,E,E,E},
  {J,J,J,J,J,J,U,J,J,J,F,J,J,J,J,J,J,X,J,J,J,U,J,J,J,J,J,J},
  {E,E,E,E,E,B,U,G,H,J,F,J,J,J,J,J,J,X,J,G,H,U,A,Z,Z,Z,Z,Z},
  {J,J,J,J,J,F,U,S,S,J,C,E,E,E,E,E,E,D,J,S,S,U,X,J,J,J,J,J},
  {J,J,J,J,J,F,U,S,S,J,J,J,J,J,J,J,J,J,J,S,S,U,X,J,J,J,J,J},
  {J,J,J,J,J,F,U,S,S,J,G,T,T,T,T,T,T,H,J,S,S,U,X,J,J,J,J,J},
  {A,E,E,E,E,D,U,W,Y,J,W,T,T,H,G,T,T,Y,J,W,Y,U,C,E,E,E,E,B},
  {F,U,U,U,U,U,U,U,U,U,U,U,U,S,S,U,U,U,U,U,U,U,U,U,U,U,U,X},
  {F,U,G,T,T,H,U,G,T,T,T,H,U,S,S,U,G,T,T,T,H,U,G,T,T,H,U,X},
  {F,U,W,T,H,S,U,W,T,T,T,Y,U,W,Y,U,W,T,T,T,Y,U,S,G,T,Y,U,X},
  {F,V,U,U,S,S,U,U,U,U,U,U,U,J,J,U,U,U,U,U,U,U,S,S,U,U,V,X},
  {O,T,H,U,S,S,U,G,H,U,G,T,T,T,T,T,T,H,U,G,H,U,S,S,U,G,T,Q},
  {P,T,Y,U,W,Y,U,S,S,U,W,T,T,H,G,T,T,Y,U,S,S,U,W,Y,U,W,T,R},
  {F,U,U,U,U,U,U,S,S,U,U,U,U,S,S,U,U,U,U,S,S,U,U,U,U,U,U,X},
  {F,U,G,T,T,T,T,Y,W,T,T,H,U,S,S,U,G,T,T,Y,W,T,T,T,T,H,U,X},
  {F,U,W,T,T,T,T,T,T,T,T,Y,U,W,Y,U,W,T,T,T,T,T,T,T,T,Y,U,X},
  {F,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,X},
  {C,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,D},
  {J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J},
  {J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J}
};
```

Le lettere usate nell'inizializzazione dell'array sono gli elementi dell'enum `oggetto` e ogni lettera rappresenta uno specifico oggetto del labirinto:

```
typedef enum {
    a='a',
    A = 'A', // ANGOLO ALTO SIN MURO ESTERNO
    B = 'B', // ANGOLO ALTO DES MURO ESTERNO
```

```
    C = 'C', // ANGOLO BASSO SIN MURO  ESTERNO
    D = 'D', // ANGOLO BASSO DES MURO ESTERNO
    E = 'E', // MURO ESTERNO OR
    F = 'F', // MURO ESTERNO VER
    G = 'G', // ANGOLO ALTO SIN
    H = 'H', // ANGOLO ALTO DES
    I = 'I', // ANGOLO ALTO DES
    J = 'J', // SPAZIO
    L = 'L', // ANGOLO ALTO SIN
    M = 'M', // ANGOLO BASSO DES
    N = 'N', // ANGOLO BASSO SIN
    O = 'O', // ANGOLO BASSO SIN
    P = 'P', // ANGOLO ALTO SIN
    Q = 'Q', // ANGOLO BASSO DES
    R = 'R', // ANGOLO ALTO DES
    S = 'S', // MURO VER
    T = 'T', // MURO OR
    U = 'U', // PUNTINO
    V = 'V', // PILLOLA
    X = 'X', // MURO ESTERNO VER
    Y = 'Y', // ANGOLO BASSO DES
    W = 'W', // ANGOLO BASSO SIN
    Z = 'Z', // MURO ESTERNO OR
} oggetto;
```

Modularizzazione del codice

Il codice del gioco è organizzato secondo il modello model-view-controller (MVC). Modello, vista e controller sono implementati in moduli separati (ad es. File). Nel file del modello c'è la matrice del labirinto, le funzioni per controllare la collisione di Pac-Man con i fantasmi e il punteggio del gioco. Le funzioni che comunicano con il video si trovano invece nel modulo di visualizzazione. Il controllo dei cicli di gioco è implementato nel modulo di controllo.
Ad ogni ciclo di gioco il controllo passa il controllo dell'esecuzione prima a Pac-Man e poi al ghost team.

L'analisi dettagliata del modello MVC è al di fuori degli

obiettivi di questo libro. Il codice completo è elencato dell'appendice "codice completo dell'MVC e dei fantasmi" e, al momento, può anche essere clonato da GitHub.

Gli agenti del Ghost Team

Gli agenti del *ghost team* sono implementati in un modulo dedicato. Ogni fantasma ha una funzione specifica che rappresenta il suo agent. Quindi, ciascun agent viene attivato dal controller chiamando la sua funzione. Queste sono:

- `gioca_blinky`
- `gioca_pinky`
- `gioca_inky`
- `gioca_clyde`

Prima di entrare nei dettagli dei singoli agenti, leggiamo questa frase del creatore di Pac-Man, che ci fa capire perché ogni fantasma ha bisogno di un agent personalizzato:

“To bring some tension into the game, I wanted the monsters to surround Pac-Man at a certain stage of the game. But I felt that it would create stress for the player if he were constantly surrounded by ghosts. Therefore, I had the monsters surround him in waves: first attack, then retreat. When they regrouped, the attack began again. It seemed to me more natural than a constant attack. ”
- Toru Iwatani, creator of Pac-Man

Toru Iwatani, ci svela quindi che le personalità del ghost team, non nascono né a caso né per pure esigenze algoritmiche, ma per creare la giusta tensione nella mente del giocatore!

Blinky , il fantasma rosso, punta direttamente alla cella di Tuki. I suoi cambi di direzione sono stabiliti solo quando attraversa un incrocio e inoltre, durante la sua caccia, non gli è permesso invertire la direzione del movimento. La strategia di Blinky lo porta ad essere il più aggressivo e temuto dei quattro.

Pinky , quello rosa, punta alla cella che si trova quattro posizioni davanti a Pac-Man. Come già specificato per Blinky, anche per Pinky e per gli altri fantasmi si applica il principio secondo cui il cambio di direzione viene preso solo in corrispondenza ad un incrocio e, per questo principio, non è escluso che Pinky catturi Pac-Man anche se punta di fronte a lui e non direttamente alla sua cella.

Inky , il fantasma blu, ha un comportamento più complesso da prevedere rispetto ai due appena visti, infatti la sua cella bersaglio non dipende solo dalla posizione di Pac-Man ma anche dalla posizione di Blinky. Inky sceglie infatti di orientarsi verso la cella intermedia tra i due.

Il comportamento di **Clyde** è un po' confuso. Inizialmente insegue Pac-Man usando la stessa strategia di Blinky, ma quando si avvicina troppo (8 celle) cambia ed entra in modalità *diffusione*, girovagando intorno ad una cella prestabilita che non è quella di Pac-Man. In questo modo Clyde aiuta a catturare Pac-Man, ma non si espone mai troppo.

N.B. La nostra implementazione degli agenti dei fantasmi si basa sulla rappresentazione del labirinto come un grafo, che è l'obiettivo principale di questo libro. Alla fine di questo libro sarete in grado di analizzare il codice di detti agenti e modificarlo a piacere.

L'agent di Pac-Man

L'agent di Pac-Man è implementato nella funzione `gioca_tuki`. Tutta la logica dell'agent deve pertanto essere implementata nella seguente funzione:

Listato `gioca_tuki_vuoto.c`

```
#include "tuki5_modello.h"
#include <stdio.h>
#include <unistd.h>

direzione gioca_tuki (posizioni posi, oggetto ** labx)
{
  direzione static ld = SINISTRA;

  return ld;
}
```

il parametro di input `posi` è una struttura che contiene le coordinate di tutti i personaggi (PAC-MAN stesso e i fantasmi):

```
  typedef struct  {
  int tuki_x, tuki_y;
  int blinky_x, blinky_y;
  int inky_x, inky_y;
  int pinky_x, pinky_y;
  int clyde_x, clyde_y;
} posizioni;
```

Tutti i nomi dei membri della struttura sono autoesplicativi tranne tuki_x e tuky_y. Il nome Tuki viene utilizzato in tutto il codice anziché Pac-Man, poiché Tuki e Giuli sono le mascottes di tutti i nostri progetti e di tutte le nostre pubblicazioni. Per questo, nella versione del nostro gioco, spesso scriviamo che il protagonista è Tuki e non Pac-Man.
Il parametro `labx` è una copia dell'array usato per rappresentare il labirinto e può essere utilizzato dall'agent

per scegliere la direzione da prendere.
Il parametro di output `ld` restituisce al controller la direzione del movimento scelta dall'agent, dove `direzione` è un tipo definito come:

```
typedef enum  {SINISTRA,SU,DESTRA,GIU,FERMO} direzione;
```

Per vincere la partita, l'agent Pac-Man deve 1) scappare dai fantasmi e 2) mangiare tutte le pillole nel labirinto. Questi obiettivi devono essere implementati nella funzione `gioca_tuki`. Questo libro ha lo scopo di presentare alcuni fondamenti della teoria dei grafi per raggiungere questi obiettivi.

Cosa è un grafo?

I grafi vengono utilizzati per modellare le relazioni tra coppie di oggetti. Ad esempio, data una serie di punti nel piano cartesiano, la distanza cartesiana tra loro può essere rappresentata con un grafo. È molto importante notare che lo stesso grafo può rappresentare molte distribuzioni diverse di punti, poiché la distanza tra i punti viene preservata ruotando l'insieme di punti nel piano.
Un famoso esempio è lo Zachary Karate Club (Zachary) in cui viene utilizzato un grafo per rappresentare le relazioni tra le coppie dei membri del club. I vertici corrispondono ai membri e gli archi che collegano due vertici rappresentano la relazione tra due membri del club.

Take home message I grafi non hanno un singolo campo di applicazione, ma possono essere usati in campi diversi per rappresentare le relazioni che possono esservi definite.

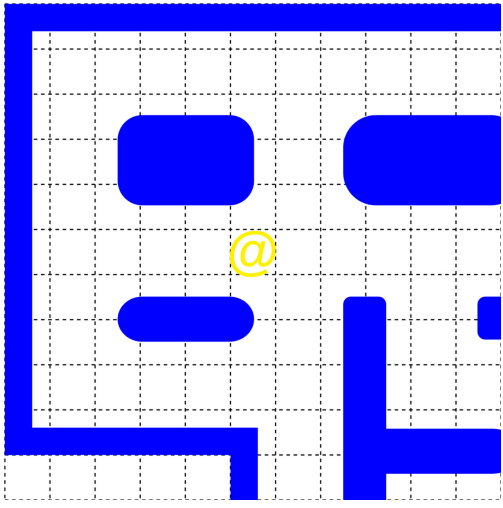
PAC-MAN AGENT LIMITATO ALLA PARTIAL OBSERVABILITY

Pac-Man vaga nel labirinto

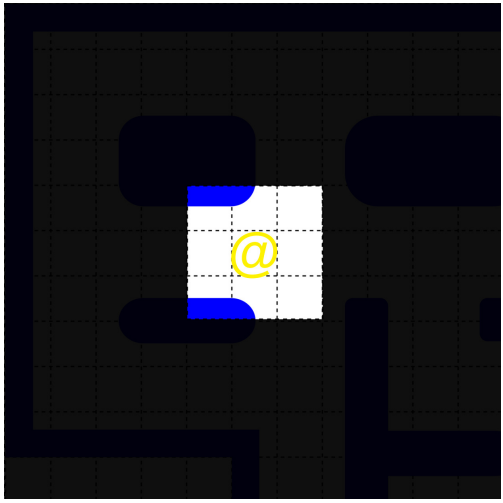
Per iniziare la nostra analisi, semplifichiamo il contesto del gioco e facciamo finta che:

1. non ci sono fantasmi a caccia di Pac-Man dentro al labirinto
2. Pac-Man abbia una Partial Observability (PO) del labirinto.

Secondo Williams (Wiliams) "PO è la compromissione della capacità di un agent di osservare completamente il mondo in cui si trova", un'idea semplice di PO è data dalle due immagini sottostanti.



1.a



1.b

Fig. 1 - La prima figura (a) mostra l'agent Pac-Man con osservabilità totale del labirinto: cioè la stessa visione del giocatore umano. Il secondo (b) mostra l'agent Pac-Man con osservabilità parziale, metaforicamente analoga alla vista ottenuta vagando con una candela in un labirinto buio.

Ad ogni ciclo di gioco Pac-Man può osservare solo le celle adiacenti alla sua posizione corrente controllando se esse appartengono a dei corridoi, quindi sono accessibili, oppure alle pareti del labirinto e quindi sono ostacoli. Ogni cellula del labirinto ha otto celle vicine.

Le celle confinanti

La riga e la colonna di Pac-Man all'interno del labirinto sono memorizzate nalla struttura `posi`

```
int i = posi.tuki_y;
int j = posi.tuki_x;
```

Ricorda: structure `posi` è un parametro passato alla funzione `gioca_tuki`

La riga e la colonna di ognuna delle otto celle confinanti con quella di Pac-Man sono ottenute a partire dalle variabili indice `i` e `j` incrementandole o diminuendole di una unità, per un totale di 2^3 otto diverse combinazioni (Vedi figura 2).

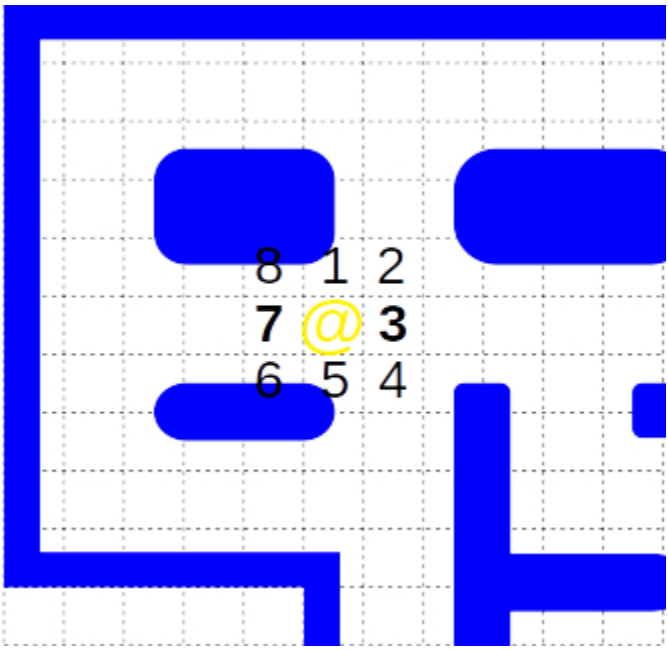


Fig. 2 - Le celle vicine (confinanti) a quella di Pac-Man sono numerate da 1 a 8. Solo le celle 3 e 7 sono accessibili, le celle 8 e 6 sono infatti occupate da elementi di muro, mentre le 2 e la 4 richiedono un movimento in diagonale che non è previsto in questo gioco. .

Per stabilire la direzione in cui muoversi, l'agent deve conoscere la disposizione dei muri rispetto alla sua attuale posizione. L'agent può memorizzare il tipo di oggetti che lo circondano *accedendo al labirinto* come segue: come segue:

```
oggetto s,d,a,b;
oggetto sa,sb,da,db;
//Ortogonali
s = labx[i][j-1]; //Sinistra
d = labx[i][j+1]; //Destra
```

```
a = labx[i-1][j]; //Alto
b = labx[i+1][j]; //BAsso
//Diagnonali
sa = labx[i-1][j-1];
sb = labx[i+1][j-1];
da = labx[i-1][j+1];
db = labx[i+1][j+1];
```

Ricorda: l'array `labx` è anch'esso un parametro della funzione `gioca_tuki`

Algoritmo dell'agent di Pac-Man

L'obiettivo di Pac-Man è di mangiare tutte le pillole distribuite lungo il campo di gioco. In questa prima fase abbiamo deciso di non considerare i fantasmi, quindi, per raggiungere il suo obiettivo, l'agent deve semplicemente attraversare in lungo e in largo tutti i corridoi a alla fine le pillole saranno esaurite.

In questa prima sezione vediamo come l'agent può percorrere tutto il labirinto, anche gli angoli nascosti, senza averne una visione dall'alto.
L'algoritmo che proponiamo si basa sulle regole seguenti:

1. non dirigerti contro i muri
2. scegli una direzione arbitraria (random) se la direzione attuale ti porta contro un muro
3. scegli una direzione casuale in corrispondenza di un incrocio
4. non cambiare direzione se non si verifica una delle condizioni precedenti.

Il `model` della piattaforma impedisce che Pac-Man attraversi un muro del labirinto indipendentemente dalla direzione di movimento scelta dall'agent (per chi ha letto *Sfidare gli algoritmi*, sempre degli stessi autori, si ricorda che il model implementa la fisica del campo di gioco.) Comunque, se l'agent sceglie male la direzione e decidere di spingere contro

un muro del labirinto, come risultato avrà perso un ciclo di gioco. Da qui si spiega la regola 1) che ha come primo obiettivo quello di evitare questo spreco.
La regola 2) risponde all'esigenza di Pac-Man di mangiare tutte le pillole e quindi di percorrere tutti i corridoi in cui esse sono presenti. Quando Pac-Man incontra un ostacolo (muro) deve scegliere una nuova direzione. Dal momento che l'agent non ha conoscenza pregressa della struttura del labirinto, per massimizzare le probabilità di percorrere tutti i corridoi, dovrà appellarsi alla statistica, confidando che una scelta casuale della direzione lo porterà prima o poi a visitare tutti i corridoi del labirinto.
Regola 3): scegliere una direzione in modo probabilistico (random) quando si incontra un ostacolo non è sufficiente a garantire il completamento del percorso, ci sono infatti nel labirinto diversi incroci che si presentano anche in corrispondenza di un cammino accessibile e non interrotto. Per questo è necessario che l'agent valuti la possibilità di cambiare direzione anche se non incontra un ostacolo.
La regola 4) infine impedisce a Pac-Man dei cambi di direzione immotivati. Questo aumenta la possibilità di percorre il labirinto in tempi brevi e evita l'effetto "ping-pong" che darebbe anche visivamente un senso di incertezza.

La condizione di PO è stata implementata nel codice che segue come già discusso poco sopra. Per implementare la regola 4), la variabile `ld`, che memorizza la direzione scelta dall'agent, è dichiarata come `static` in modo da poterne confrontare il valore nei successivi turni di gioco.

Limiti della soluzione implementata

La soluzione implementata nel codice visto sopra è una soluzione *state-less*, cioè, fatto salvo per la direzione scelta,

non preserva lo stato di conoscenza acquisita dall'agent tra un turno di gioco e il successivo.
L'approccio state-less può essere una soluzione efficiente in alcune specifiche topologie di labirinto. Per esempio si consideri un labirinto alternativo a quello di Pac-Man costituito da un semplice otto, cioè due percorsi ciclici che si uniscono in un singolo incrocio.
È immediato realizzare che anche un algoritmo state-less può garantire il completamento di tale percorso in tempi certi è infatti sufficiente impostare l'agent in modo che in caso di incrocio scelga (per esempio) la direzione destra, e in questo modo all'incrocio tra i due anelli l'agent abbandona il primo e segue il secondo.

```
ld = DESTRA;
```

La stessa logica applicata al labirinto di Pac-Man lo porterebbe a girare ininterrotto attorno ad una delle aiuole senza più completare il resto del percorso.

Prima di discutere come migliorare l'agent del Pac-Man, è il momento di provarne l'implementazione e vederla in pratica.

Compilare il codice e lanciare l'eseguibile

Iniziamo a vedere come compilare ed eseguire i sorgenti. Per questa prima prova implementiamo per Pac-Man un agent molto basilare che lo faccia rimanere fermo all'interno del labirinto.

Ci sono due modi per compilare il codice: 1) editare il listato, 2) scaricare o clonare il progetto da GitHub.
Per il modo 1):

1. Creare la struttura di directory seguente:
    - `grafi`

- grafi/ghost_team
- grafi/mvc
- grafi/PacMan

2. creare un file nominato grafi/compila.sh con il codice seguente:

   Listato compila.sh

```
#!/bin/sh

if [ "$#" -ne 3 ]; then
    echo "3  parameters required:\n
    \tUser codice file (gioca_tuki.c for example)\n
    \t1 or 0 to activate or no run away option\n
    \tTime delay for each game cycles (ms)\n"
    exit
fi

gcc -pedantic -std=c99 -o tuki5.game -DFUGA=$2 -DDELAY=
$1 \
ghost_team/gioca_fantasmi.c \
mvc/tuki5_controllo.c \
mvc/tuki5_modello.c \
mvc/tuki5_visore.c \
-Imvc \
-lm
```

3. Editare i file seguenti:
   - grafi/mvc/tuki5_modello.h
   - grafi/mvc/tuki5_modello.c
   - grafi/mvc/tuki5_visore.h
   - grafi/mvc/tuki5_visore.c
   - grafi/ghost-team/gioca_fantasmi.c
   - grafi/PacMan/gioca_tuki_vuoto.c
4. Cambiare la directory in grafi e lanciare compila.sh:
   - ~\$ cd grafi
   - ~/grafi\$ chmod +x compila.sh
   - ~/grafi\$ ./compila.sh PacMan/gioca_tuki_vuoto.c 0 100

   I tre parametri passati allo script (compila.sh) sono: il file utente con il codice dell'agent che si vuole integrare alla piattaforma, un booleano (0 o 1) che determina se

attivare l'opzione di fuga dell'agent dal ghost-team, il ritardo in millisecondi con cui viene mostrato ogni turno di gioco.

5. Imposta il terminale su almeno 50 righe e 50 colonne e avvia l'eseguibile `tuki5.game` per iniziare il gioco:
    - `~/grafi\$ ./tuki5.game`

    All'avvio del programma si deve vedere sul terminale lo stesso che è riportato in figura 3. Si ricordi però che il modulo `viewer` non modifica il colore di sfondo del monitor, quindi se si vuole che sia nero lo si setti dalle proprietà della finestra, o si setti qualsiasi altro colore compatibilmente con i colori che sono già stati usati nel gioco.

    Come si vedrà, Pac-Man rimane fermo al centro del labirinto mentre i fantasmi del ghost team lo circondano e lo bloccano (mangiano). Pac-Man non prova a fuggire né a muoversi perché il codice dell'agent che abbiamo inserito come input a `compila.sh` non ha logica di fuga né di esplorazione del labirinto.

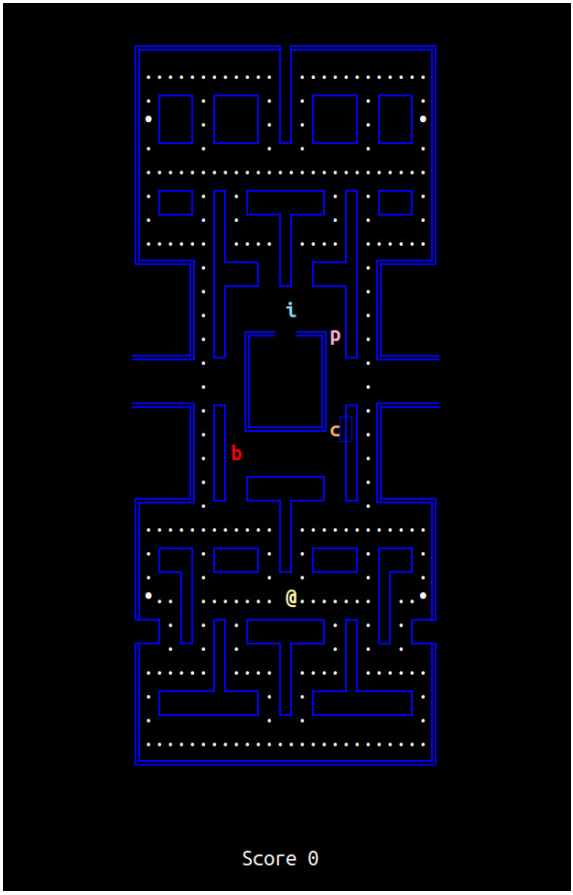


Fig. 3 - Il labirinto di Pac-Man creato come una griglia di caratteri.

Lanciare `gioca_tuki_random.c`

A questo punto si è capito lo scopo di ogni file della piattaforma e il ruolo dello *user code* `gioca_tuki_(quello che vuoi).c`. Ora è il momento di provare il codice implementato nell'agent *random* e vedere cosa è capace di fare!

Si seguano le istruzioni seguenti:

```
~/grafi\$ ./compila.sh PacMan/gioca_tuki_random.c 0 50
~/grafi\$ ./tuki5.game
```

Dopo l'avvio del programma Pac-Man inizia a vagare per il labirinto seguendo le quattro regole viste prima. Ai fantasmi occorrerà poco tempo per catturarlo perché non è stata attivata l'opzione di fuga.
Per vedere Pac-Man completare il labirinto specifichiamo che non ci devono essere fantasmi nel ghost team e lanciamo ancora l'esecuzione:
`~/grafi\$ ./tuki5.game 0`
Lo 0 indica il numero di fantasmi in azione.
A seconda del valore DELAY usato durante la compilazione, si dovranno aspettare un paio di minuti affinché Pac-Man finisca il labirinto. Guardando questa gara con zero fantasmi si potrà capire la personalità dell'agent codificato nel file sorgente e iniziare a pensare a come migliorarlo!
Il `model` non impedisce all'agent di entrare nella casa del ghost team, dove Pac-Man può anche essere catturato, specialmente se i quattro fantasmi sono all'interno. Se si preferisce impedire questa eventualità è necessario modificare il labirinto con il codice seguente:

```
labx[16][2]  = 'A';
labx[16][23] = 'A';
labx[14][13] = 'A';
labx[14][14] = 'A';
```

La fuga di Pac-Man

Fin'ora si è visto come implementare un semplice agent che basandosi sulla eventualità probabilistica ha ottime prospettive di completare il labirinto, però come si è potuto verificare la sua esplorazione ha vita breve se si abilitano uno o più fantasmi del ghost team.

Analizzando il codice `gioca_tuki_random.c` si vede però che è stata implementata una logica che permette a Pac-Man di evitare gli attacchi diretti dei fantasmi.

```
int x = posi.tuki_x;
int y = posi.tuki_y;
int x_g[4];
int y_g[4];
...
char s_g = 0, d_g = 0,a_g = 0,b_g = 0;
for (int ig = 0; ig<4; ig++)
  {
    s_g = s_g || ( x_g[ig] < x) &&
     ( x - x_g[ig] <=2 )  && (y == y_g[ig]);
    s_g = s_g || ((x_g[ig] == x-1) &&
      (y_g[ig] == y+1));
    s_g = s_g || ((x_g[ig] == x-1) &&
      (y_g[ig] == y-1));
    d_g = d_g || ( x_g[ig] > x) &&
     (x_g[ig] - x <= 2) && (y == y_g[ig]);
    d_g = d_g || ((x_g[ig] == x+1) &&
      (y_g[ig] == y+1));
    d_g = d_g || ((x_g[ig] == x+1) &&
      (y_g[ig] == y-1));
    a_g = a_g || (x == x_g[ig]) &&
     ( y >  y_g[ig]) && ( y - y_g[ig] <=2);
    a_g = a_g || ((y == y_g[ig] + 1) &&
     (x_g[ig] == x+1));
    a_g = a_g || ((y == y_g[ig] + 1) &&
     (x_g[ig] == x-1));
    b_g = b_g || (x == x_g[ig]) &&
     ( y <  y_g[ig]) && ( y_g[ig]- y <=2);
    b_g = b_g || ((y == y_g[ig] - 1) &&
     (x_g[ig] == x+1));
    b_g = b_g || ((y == y_g[ig] - 1) &&
     (x_g[ig] == x-1));
  }
  ...
  if((s_g || d_g || a_g || b_g) && FUGA)
  {
  ...
  }
```

In pratica viene verificato che le coordinate di ogni fantasma (ciclo `for` da 1 a 4) non siano adiacenti a quelle di Pac-Man e, in questo caso, viene impostato a 1 il flag di prossimità che permette all'agent di cambiare direzione rispetto a quella di cattura.

Per provare la modalità fuga, si compili impostando ad 1 il flag di fuga (vedi sopra) e si lanci l'eseguibile specificando 1 come numero di fantasmi.

```
~/grafi\$ ./tuki5.game 1
```

Box domande n.1

1. Cos'è la Partial Observability?
    a. L'incapacità dell'agent di vedere oltre le celle a lui immediatamente adiacenti
    b. L'incapacità dell'agent di vedere in diagonale
    c. La possibilità andare in una sola direzione
4. Se un fantasma è a due celle di distanza dal Pac-Man e Pac-Man è giudato dall'agent *random*, Pac-Man cambia direzione?
    a. Sì, perché l'agent random tende a far stare il Pac-Man a distanza di sicurezza dai fantasmi
    b. No, perché l'algoritmo di fuga prevede un avvicinamento al fantasma prima di scappare
    c. No, perché la partial observability gli impedisce di vedere a due celle di distanza
4. Esiste la possibilità che il Pac-Man, guidato dall'agent *random*, non completi mai il labirinto?
    a. Sì, la statistica ci dice che in un tempo infinito il completamento è certo, ma nel contesto di un'esecuzione che avviene in un tempo limitato, può sempre succedere
    b. Sì, perché si tratta di un moto aleatorio
    c. No

Pac-Man attraversa in modo deterministico tutto il labirinto

Nei paragrafi precedenti abbiamo visto come l'agent sia capace di completare il labirinto e mangiare tutte le pillole, basandosi solo sulla teoria delle probabilità o, in parole più semplici, scegliendo a caso dove svoltare ad ogni incrocio.
In molte situazioni questo approccio non è la soluzione ottimale, anche per le seguenti ragioni:

- il metodo probabilistico da risultati garantiti solo in un tempo virtualmente infinito
- anche nel caso che l'agent completi il labirinto potrebbe impiegarci un tempo inutilmente lungo.

Si immagini di lavorare in un'agenzia spaziale e di inviare un rover su Marte per esplorare l'intero territorio di una regione specifica e restituire alcuni dati di interesse. Sarebbe logico implementare l'algoritmo di *vagabondaggio* probabilistico come agent del rover?
Probabilmente no, meglio continuare l'analisi e scoprire algoritmi più efficienti.

La pianificazione della *copertura* completa di ambienti parzialmente noti o completamente sconosciuti è un problema comune nelle applicazioni di robotica anche sulla Terra, ad esempio per l'agricoltura assistita robotizzata. È un dato di fatto che questo tipo di attività è diventata molto comune con l'avvento dei robot mobili e dei micro-aerei (droni). Ad esempio, ci sono robot mobili che devono tagliare

l'erba di un campo evitando di rovinare le aiuole dei fiori o di altre piante ornamentali, mentre per quel che riguarda i droni, questi devono definire delle *no flight zone* per evitare ostacoli come alberi o edifici.
Per ora quindi, lascia che Marte ci aspetti e proseguiamo con l'analisi degli algoritmi.

Decomposizione cellulare

In questa sezione presenteremo il metodo detto *boustrophedon*. Questo è un algoritmo specifico per eseguire una *decomposizione cellulare* di un'*area di lavoro*.

Con il termine area di lavoro si intende l'ambiente, la zona o lo spazio in cui l'agent deve agire. Nel caso di robot reali, l'area di lavoro potrebbe essere un giardino un cui tagliare l'erba, mentre per i droni potrebbe essere lo spazio sopra una città o tra i palazzi. Nel caso del Pac-Man sarà il suo labirinto.

Questo algoritmo è ampiamente utilizzato nelle odierne applicazioni di pianificazione di percorsi robotizzati (Bähnemann).

La sfida principale nella pianificazione automatica della copertura di un ambiente è la gestione degli ostacoli.

Con copertura si intende l'esplorazione completa dell'ambiente.

Per essere indipendente, l'agent di un robot dovrebbe essere in grado di identificare un percorso che lo conduca da un punto iniziale a un punto finale evitando collisioni.
Matematicamente, il problema può essere rappresentato come segue:

- Sia A un robot che si sposta in un'area di lavoro W e sia W un sottoinsieme di R^2
- W è partizionato in regioni C_{libera} e $C_{ostacolo}$, che sono

rispettivamente regioni disponibili e non disponibili
- dato un punto iniziale e un punto finale, l'obiettivo è quello di trovare un percorso che li colleghi e che si trovi completamente sulla regione C_{libera}.

Esistono diversi metodi che possono risolvere questo problema, qui ci concentreremo sulla decomposizione cellulare.

Dato un ambiente, il metodo di decomposizione cellulare è il primo passo per raggiungere la pianificazione e la copertura del percorso. Si compone di due passaggi:

1. dividere l'ambiente in celle
2. trasformare l'ambiente in un grafo in cui i vertici sono le celle e gli archi collegano i vertici corrispondenti alle celle adiacenti

Nelle applicazioni della vita reale, il passaggio 1 può essere una vera sfida da realizzare, per fortuna il labirinto Pac-Man è già composto da celle uniformi, quindi questo passaggio risulterà più semplice.

La decomposizione cellulare divide C_{libera} in un insieme finito di regioni (celle) in modo tale che:

1. è triviale calcolare un percorso privo di collisioni tra due punti della stessa cella
2. è facile scoprire i vicini di ogni cella per costruire la roadmap (percorso)
3. è possibile determinare per ogni punto quale sia la cella che lo contiene

Nella figura 5 mostriamo l'esatta decomposizione cellulare di uno spazio di lavoro mostrato nella figura 4. L'area verde (grigio chiaro) rappresenta C_{libera}, mentre i due poligoni

arancioni (grigio scuro) sono gli ostacoli.

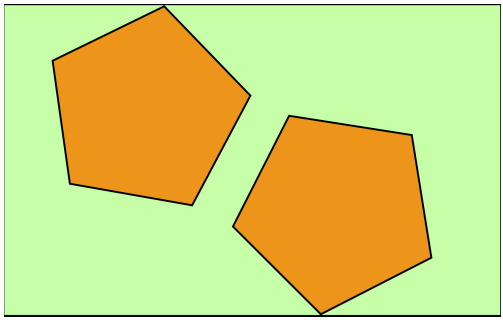


Fig. 4 - Un esempio di un ambiente diviso in regioni accessibili (verde o grigio chiaro) e inaccessibili (arancione o grigio scuro).

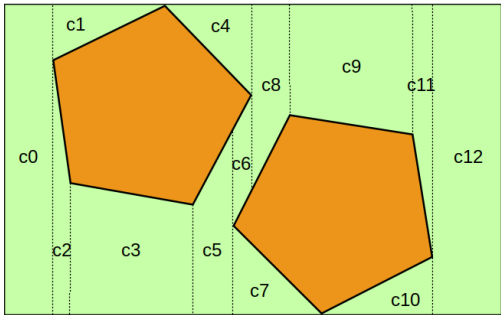


Fig. 5 - L'ambiente dopo la decomposizione cellulare.

Una volta effettuata la decomposizione delle celle, è possibile mappare C_{libera} in un grafo (figura 6) in cui:

- ogni cella è un vertice
- i vertici corrispondenti alle celle adiacenti sono collegati da un bordo.

Una volta che lo spazio di lavoro è stato mappato su un grafo, il problema di coprire tutte le celle accessibili diventa un *problema di grafi* (a graph problem) che può potenzialmente essere risolto con algoritmi su grafi esistenti.

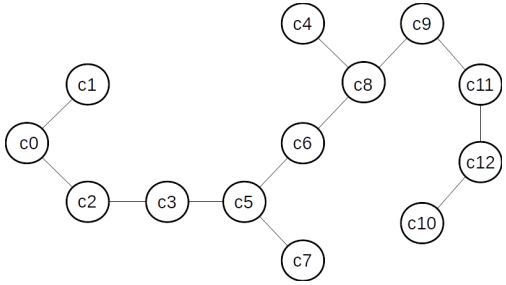


Fig. 6 - Il grafo creato dopo la decomposizione cellulare dell'area di lavoro.

Take home message: Lo spazio di lavoro può essere mappato in un grafo in modo che l' *attività di copertura* venga trasformata in algoritmi sui grafi.

Boustrophedon algorithm

L'algoritmo è stato descritto la prima volta da Choset (Choset). L'idea è di iniziare il percorso da una cella detta appunto cella iniziale per poi continuare a spostarsi da una cella all'altra in una determinata direzione che verrà chiamata direzione preferenziale.
La direzione intrapresa ad ogni passaggio di cella viene aggiunta a un elenco di passaggi che verranno utilizzati quando verrà rilevato un vicolo cieco nell'area di lavoro. Ogni cella in cui entra l'agent viene contrassegnata come visitata.
L'agent continua a spostarsi nella direzione preferenziale fino a quando non incontra un ostacolo. In tale situazione, controlla l'accessibilità di una cella vicina diversa dal suo obiettivo. Questo viene fatto cambiando direzione e cercando di fare un passo avanti. Se viene trovata una direzione accessibile, l'agent fa il passo in quella direzione quindi l'esecuzione dell'algoritmo continua come definito sopra. Al contrario, se l'agent incontra il vicolo cieco e quindi non è disponibile alcuna cella adiacente, fa un passo indietro e controlla le celle accessibili che lo circondano e che non siano

già state visitate.

Nella figura sotto abbiamo disegnato un labirinto semplificato ritagliato dal labirinto originale. Pac-Man cammina attraverso il perimetro rettangolare maggiore evitando di prendere il primo incrocio sulla destra (linea verde o grigio chiaro). Quando raggiunge la posizione iniziale, scopre che non ci sono più celle non visitate intorno a lui, quindi torna indietro (linea rossa o grigio scuro) fino a raggiungere l'intersezione e prende la direzione che non aveva preso la prima volta (linea verde o grigio chiaro).

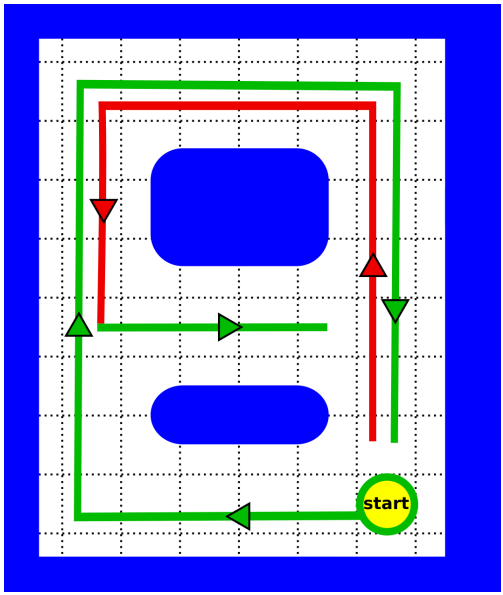


Fig. 7 - Pac-Man torna indietro quando non trova più celle disponibili.

Strutture dati

Per rappresentare ogni cella nel labirinto di Pac-Man useremo il tipo `struct agri_Cella` definito come di seguito.

Ricorda di leggere l'appendice per comprendere il significato dei nomi e dei verbi latini.

```
typedef struct cella {
  Attributi d;
  struct cella* dextra;
  struct cella* deorsum; //Giu
  struct cella* sinistra;
  struct cella* sursum;  //Su
} agri_Cella;
```

La struttura contiene quattro campi che sono dei puntatori alle quattro `agri_Cella` sue vicine, e un campo che descrive i suoi attributi definiti nella seguente struttura:

```
typedef struct dato
{
  int visitata;
  rei_genus rei;
} Attributi;
```

L'attributo `visitata` è 0 se la cella è stata rilevata durante la passeggiata ma non visitata, mentre è 1 se la cella è stata anche visitata. Si noti che se la cella non è stata rilevata, allora non esiste ancora nel grafo dell'agent.
L'attributo `rei_genus` può essere "muro" o "qualcos'altro" e indica cosa contiene la cella.

Take home message: ogni cella può essere pensata come un vertice di un grafo in cui i bordi sono le connessioni tra le celle.

Connessione tra i vertici del grafo

Nell'ultimo *take home message*, abbiamo realizzato che le celle del labirinto sono come i vertici di un grafo. Tali vertici vengono scoperti durante l'esplorazione dell'agent e devono essere collegati tra loro. Questo compito viene implementato facendo in modo che le due celle (cella di Pac-Man e cella scoperta) puntino l'una verso l'altra nella direzione in cui è stata trovata la cella. Questo viene fatto usando la seguente funzione:

```
agri_Tabella agri_addo_Tabellam(agri_Tabella tabella,Attribu
{
```

```
  agri_Cella* n=(agri_Cella*)malloc(sizeof(agri_Cella));
  n->d=d;

  switch(dir)
    {
    case DEORSUM:
      tabella->sursum=n;
      n->deorsum=tabella;
      break;
    case SURSUM:
      tabella->deorsum=n;
      n->sursum=tabella;
      break;
    case DX:
      tabella->dextra=n;
      n->sinistra=tabella;
      break;
    case SX:
      tabella->sinistra=n;
      n->dextra=tabella;
      break;
    }
  return n;
}
```

dove `agri_Tabella` è definito come:

```
typedef agri_Cella* agri_Tabella;
```

Come si può notare, questo tipo di grafo è una sorta di tabella, infatti ogni vertice ha esattamente quattro spigoli.
L'idea di base è quella di utilizzare questa rappresentazione del grafo per creare l'algoritmo boustrophedon che consente a Pac-Man di attraversare il labirinto mantenendo una mappa costantemente aggiornata.

Spostamento verso la cella successiva

Ad ogni passo, Pac-Man collega fino a quattro vertici alla sua cella facendo crescere il grafo. Una volta che Pac-Man ha collegato tutte le celle vicine alla sua, deve decidere la cella successiva in cui spostarsi (cella bersaglio). Partendo da sinistra (la direzione preferenziale), l'agent controlla le altre tre direzioni, in ordine: su, destra e giù, fino a quando non

trova una cella che non sia stata visitata e che non sia occupata da un muro.

```
static agri_Tabella g=NULL;
...
if(g->sinistra->d.visitata == 0
&& g->sinistra->d.rei != MURO)
   {
     g=g->sinistra;
     g->d.visitata=1;
     agri_addo_Iter(&p,g);
     l=p;
     return SINISTRA;
   }
```

Se alla fine l'agent non ha trovato una cella disponibile, inizia a camminare all'indietro per trovare una cella che presenti almeno un incrocio inesplorato.

```
/* Retraces his steps */

 l=l->prev;
 if(l==NULL) exit(-1);

 agri_addo_Iter(&p,l->locus);

 direzione nd;

 if(g->dextra==l->locus) nd=DESTRA; //Right
 else
 if(g->sinistra==l->locus) nd=SINISTRA; //Left
 else
 if(g->sursum==l->locus) nd=SU; //Up
 else
 if(g->deorsum==l->locus) nd=GIU; //Down
```

La traccia del cammino percorso

Dopo aver letto l'ultimo pezzo di codice ci si può chiedere cosa sia `l` . Questa variabile è definita come:

```
static agri_Iter l = NULL;
```

dove:

```
typedef agri_Passo* agri_Iter;
```

e

```
 typedef struct nodo
{
  agri_Cella * locus;
  struct nodo * next;
  struct nodo * prev;
}agri_Passo;
```

La variabile `l` punta ad una lista collegata contenente i passaggi (passi) che Pac-Man ha effettuato fino alla cella corrente, indicata dalla variabile `g` .

La libreria `agri`

Prima di presentare l'algoritmo boustrophedon completo, mostriamo il codice di una libreria che abbiamo creato esplicitamente per gestire grafi e gli algoritmi per questo testo.
Sebbene le funzioni della libreria siano limitate agli argomenti trattati in questo libro, la comprensione della sua logica e l'abitudine ad usarla è propedeutica a quelle professionali come igraph.

Ci si può chiedere perché questa libreria si chiama `agri`.
Il codice qui presentato deve gestire l'agent Pac-Man in movimento nel proprio *campo* di gioco, cioè il labirinto. Da questo nasce il nome Ager che significa appunto campo con l'accezione che ha campo in "campo di fiori" e in "campo di battaglia".
Per ricordare facilmente i nomi dei tipi e delle funzioni, consigliamo vivamente di leggere l'appendice: basi del latino.
In appendice sono riportati i codici sorgenti della libreria, che andranno editati dopo aver creato l'opportuno albero di directory come spiegato nel paragrafo che segue.

Preparare la compilazione `libagri`

1. Si crei il seguente albero di directory:
    - `grafi/agri/lib`

- grafi/agri/lib/src
- grafi/agri/lib/include

2. Si crei il file grafi/compila.sh come segue:

   Listato compila.sh

   ```
   #!/bin/sh

   if [ "$#" -ne 3 ]; then
       echo "3  parameters required:\n
       \tUser code file (gioca_tuki.c for example)\n
       \t1 or 0 to activate or no run away option\n
       \tTime delay for each game cycles (ms)\n"
       exit
   fi

   gcc -pedantic -std=c99 -o tuki5.game -DFUGA=$2 -DDELAY=
   $1 \
   ghost_team/gioca_fantasmi.c \
   mvc/tuki5_controllo.c \
   mvc/tuki5_modello.c \
   mvc/tuki5_visore.c \
   agri/lib/src/libagri.c \
   -Iagri/lib/include \
   -Imvc \
   -lm
   ```

   dove sono state aggiunte le istruzioni

   ```
   agri/lib/src/libagri.c \
   -Iagri/lib/include \
   ```

   allo script compila.sh

3. Editare e salvare i file come segue:
   - grafi/agri/lib/src/libagri.c
   - grafi/agri/lib/include/libagri.h

L'agent boustrophedon

Questa sezione presenta il codice dell'agent boustrophedon. Questo fornisce a Pac-Man la capacità di attraversare il labirinto costruendone la rappresentazione in un grafo dalle celle su cui si è mosso.
Questo grafo viene creato al momento del gioco e utilizzato dall'agent come mappa per avere un metodo algoritmico per

coprire l'intero labirinto.

Si editi il codice e lo si salvi nel file:

`grafi/PacMan/gioca_tuki_boustrophedon.c`

Si compili quindi il codice con le opzioni seguenti:

```
~/grafi\$ ./compila.sh
PacMan/gioca_tuki_boustrophedon.c 0 50
```

Si esegua quindi il programma escludendo la presenza del ghost team:

```
~/grafi\$ ./tuki5.game 0
```

Eseguendo il codice più volte, si può notare che partendo dalla stessa posizione, il percorso di Pac-Man sarà sempre lo stesso. Questo significa che l'agent ha un metodo per completare l'intero labirinto che non si basa su decisioni casuali, che è esattamente l'obiettivo di questa sezione.

Potete verificarlo provando a cambiare la posizione iniziale di Pac-Man nel `model`, avendo cura di posizionarlo lungo un corridoio e non all'interno di un muro. Vi accorgerete che ad ogni posizione iniziale corrisponde un preciso percorso, che sarà lo stesso ogni volta che lancerete il gioco.

Boustrophedon e ghost team

Nei paragrafi precedenti si è analizzata la sezione di codice volta a far fuggire Pac-Man dai fantasmi.
Questa strategia di fuga si basa sulla Partial Observability che stiamo simulando, infatti, ad ogni ciclo di gioco, l'agent controlla se ci sono fantasmi solo nelle celle vicine.

Ora attiviamo il codice di escape impostando il secondo parametro su 1 nello script della shell:

```
~/grafi\$ ./compila.sh
```

```
PacMan/gioca_tuki_boustrophedon.c 1 50
```

e lanciamo di nuovo l'eseguibile specificando la presenza di un solo fantasma:

```
~/grafi\$ ./tuki5.game 1
```

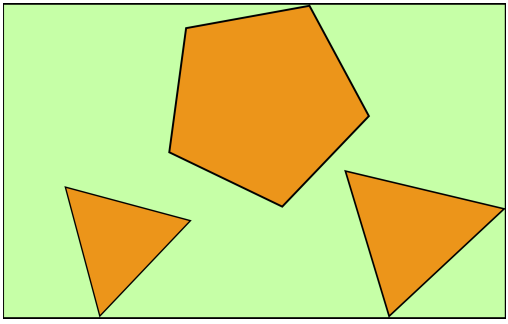
Un fantasma da solo non può catturare Pac-Man ma, come si vede, i due cadranno in un ciclo di fuga infinito, e questo è dovuto all'eccessiva rigidità del boustrophedon.
Una volta provato con un solo fantasma si può provare ad attivare due, tre e quattro fantasmi.

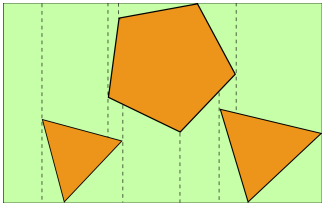
Take home message: Un grafo può essere usato come rappresentazione del labirinto di Pac-Man per fornire al suo agent un metodo per coprirlo e cercare di sfuggire ai fantasmi senza perdersi.

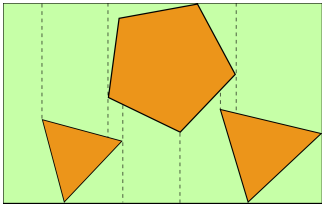
In questa sezione è stato presentato e approfondito il concetto di rappresentazione dello spazio di lavoro all'interno dell'agent e si è visto come su tale rappresentazione può *lavorare* un algoritmo per assicurare un compito, nel caso specifico la copertura completa dell'area di lavoro.
È interessante approfondire questo concetto confrontandolo con l'approccio quasi opposto del lavoro di Brooks riguardo alla realizzazione di robot che interagiscono con l'ambiente senza crearsene una rappresentazione in memoria, quindi più simile all'agent state-less visti in precedenza (Brooks). Se ne consiglia la lettura.

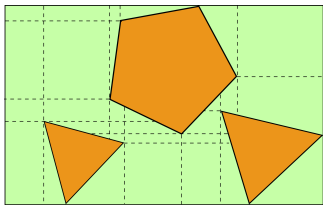
Box domande n.2

1. Quale tra queste affermazioni sulla Decomposizione Cellulare è falsa?
    a. La superficie appartenente agli ostacoli non è considerata come cella
    b. A volte è impossibile calcolare un percorso tra due punti appartenenti alla stessa cella
    c. E' possibile ricavare un grafo della decomposizione cellulare dell'ambiente
4. Dato il seguente campo di lavoro, quale tra le tre decomposizione cellulari è quella corretta?

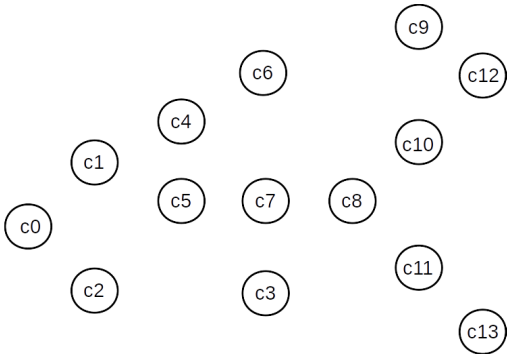


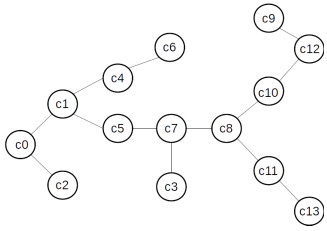
a. 

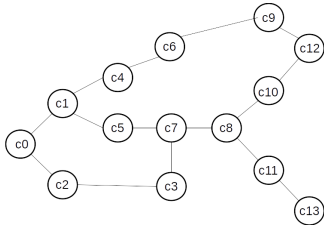
b. 

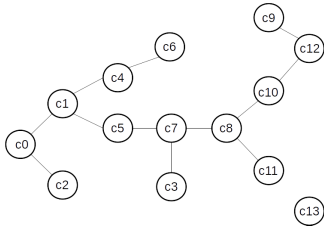
c. 

4. Come sono collegati i vertici del grafo ottenuto dalla decomposizione cellulare del campo di lavoro della domanda precedente?



a. 

b. 

c. 

Creazione di un grafo

Grafo con numero variabile di archi

Nell'informatica, i grafi sono implementati come strutture dati e funzioni che agiscono su queste.
Esistono diversi modelli di grafi che sono adatti a diversi scenari. Si può scegliere il modello giusto per uno scenario d'uso specifico al fine di semplificare l'implementazione o il riutilizzo del codice.
Nell'algoritmo boustrophedon il grafo è stato implementato da una matrice di strutture ognuna delle quali rappresenta un vertice del grafo. Ora, sebbene il labirinto rimane lo stesso, si propone un'implementazione diversa, basata più sugli archi che non sui vertici. Vediamo come.

Affrontiamo il problema della costruzione di un grafo che abbia un numero variabile di archi per ciascun vertice. Il labirinto di Pac-Man infatti può anche essere pensato come un labirinto di corridoi che si intersecano. Usando questo modello, un grafo più rappresentativo è quello in cui le intersezioni tra i corridoi (incroci) sono i vertici del grafo e i corridoi stessi sono gli archi.

Si confronti questo modello con quello usato nell'esempio del boustrophedon dove abbiamo usato un grafo che rappresentasse tutte le celle della tabella (matrice) del labirinto, ognuna collegata alle sue quattro celle vicine.

Per creare un tale grafo possiamo analizzare il labirinto partendo da un incrocio (intersezione) che rappresenterà il primo vertice, e aggiungere ad esso degli archi per ogni corridoio presente nell'incrocio. Quindi, seguiamo il corridoio aggiunto fino al prossimo incrocio, e questo sarà il secondo

vertice del grafo. Ripetendo questa procedura per ogni corridoio, copriremo l'intero labirinto ottenendo il grafo del labirinto di Pac-Man.
Come esseri umani non abbiamo bisogno di un algoritmo formale per farlo, siamo semplicemente in grado di farlo. In realtà, il compito diventerebbe difficile se avessimo un labirinto molto grande e in una situazione del genere avremo bisogno di un algoritmo da seguire per non perderci.
In questa sezione insegnamo all'agent Pac-Man come creare un grafo "Vertici e archi" durante il gioco. Poiché il grafo non è noto all'agent all'inizio del gioco (ha bisogno di percorrere il labirinto per crearlo), stiamo ancora studiando lo scenario PO.

L'agent

L'obiettivo è fare in modo che l'agent generi un grafo del labirinto usando osservazioni online (tempo reale).
In ogni ciclo di gioco Pac-Man può solo osservare le celle adiacenti alla sua posizione corrente verificando se appartengono a corridoi accessibili o pareti del labirinto. Ogni cella del labirinto ha otto celle vicine. Per generare il grafo, Pac-Man dovrebbe considerare ciascuna di esse, comunque sappiamo che l'architettura a labirinto non consente movimenti obliqui, quindi possiamo limitare l'agent Pac-Man a controllare solo le celle vicine nelle direzioni su, destra, giù e sinistra.
Pac-Man classifica come appartenente a un arco ogni cella che ha solo due vicini accessibili e come vertice quelle che ne hanno tre o più.

Strutture dati

Dalla teoria sappiamo che un grafo può essere rappresentato come un insieme di vertici e un insieme di archi i quali

collegano ciascuno una coppia di vertici (non più di due).

Come mostrato nel codice seguente, un arco è rappresentato attraverso la struttura (`agri_Colligatio`) ed ha due vertici (`agri_Vertex`) come la sua definizione formale dalla teoria dei grafi.

```
typedef struct colligatio {
  agri_Vertex ab, ad;
  versus discessus, meta;
  int longitudo;
} agri_Colligatio;
```

La teoria dei grafi non specifica cosa sia un vertice perché ciò dipende da quello che si vuol rappresentare con esso. Ad esempio, un vertice potrebbe essere una persona, come abbiamo visto nel famoso esempio del Zachary's karate club, oppure potrebbe essere uno scaffale di un magazzino, ma anche solo un indice numerico astratto.
Di seguito, rappresentiamo i vertici come tipi `struct` per poter aggiungere loro attributi di riga e colonna oltre al loro indice che li identifica nel labirinto.

```
typedef struct {
  int linea;
  int columna;
  int index;
  int ianua[PORTE];
} agri_Vertex;
```

Non è necessario creare un file con queste `struct` perché sono già presenti nei file di libreria `libagri` .

Nel codice in appendice `gioca_tuki_generagrafo.c`, Pac-Man costruisce il grafo mentre vaga per il labirinto. Ad ogni ciclo di gioco classifica la sua cella come un vertice o parte di un arco. La decisione viene presa contando il numero di celle vicine accessibili. Se ce ne sono due, Pac-Man si trova su un arco del grafo, altrimenti se ce ne sono più di due, Pac-Man

si trova su un vertice come mostra il codice *semplificato* di seguito.

```
int nd = 0;
for(int k=0; k<4; k++)
  nd += (1*oggetto_accessibile(vicino[k]));


/*
   ITA: È vero se nel ciclo di gioco corrente viene rileva
   ENG: Is true if in the current game cycle a vertex is d
*/
bool nodo_rilevato = false;
if(nd>2)
  {
    /*
       ITA: Se siamo qui, Tuki è su un vertice
       ENG: If we are here Tuki position is a vertex
    */
    int vertice_a = agri_Verticem_quaero(g,i,j);
    if(vertice_a<0)
      {
        vertice_a = vertici_contati;
        vertici_contati++;
      }
  }
```

Take home message:in C, un grafo può essere implementato con una lista collegata i cui elementi sono gli archi.

Compilazione ed esecuzione

Si compili il codice con le opzioni specificate nel seguito:

```
~/grafi\$                                  ./compila.sh
PacMan/gioca_tuki_generagrafo.c 0 50
```

Si lanci specificando nessun fantasma:

```
~/grafi\$ ./tuki5.game 0
```

Ciò che si vedrà nel terminale potrebbe sorprendere perché nulla è cambiato dall' esecuzione del codice sviluppato per

l'agent random.
Però questa è solo metà della storia, bisogna continuare a leggere per scoprire l'altra metà.

Come sarà il grafo prodotto?

Nel labirinto di Pac-Man, ogni arco è un corridoio che collega direttamente due incroci. Poiché ci sono 34 intersezioni nel labirinto, il grafo risultante avrà 34 vertici.

Questo è diverso dalla rappresentazione del boustrophedon, in cui ogni cella del labirinto è un vertice collegato da archi *virtuali* alle sue quattro celle vicine, il tutto risultante in un numero maggiore di vertici nel grafo.

Per avere una rappresentazione grafica del grafo, vediamo che un grafo può essere rappresentato come una matrice quadrata i cui indici di colonna e riga sono i vertici. Costruiamo la matrice in modo che ogni elemento valga 1 o 0.

Agli elementi della matrice può essere assegnato anche un valore non binario. Per esempio, se si sta rappresentando un grafo delle connessioni stradali tra le città, si potrebbe assegnare agli elementi della matrice la distanza tra le due città individuate dalla riga e dalla colonna della matrice stessa.

L'elemento nella riga *i* e colonna *j* è diverso da 0 se esiste un bordo che collega i vertici *i* e *j* (vedi fig. 8)

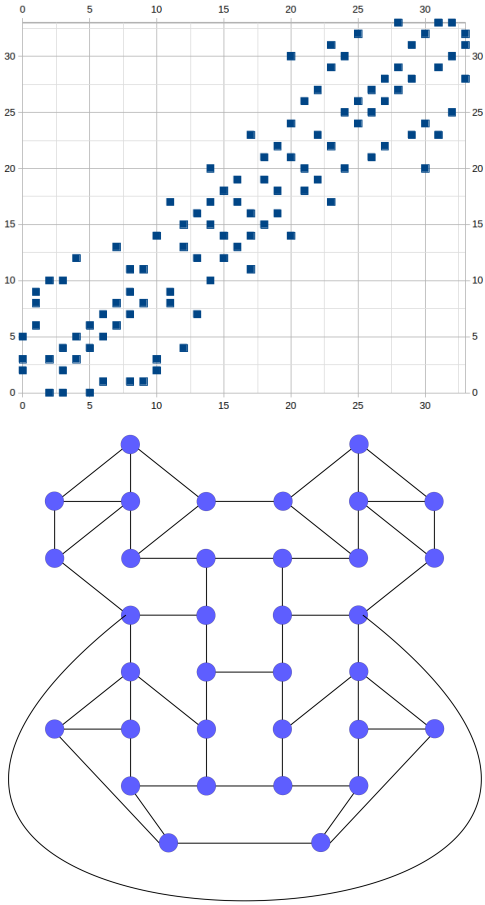


Fig. 8 - Rappresentazione matriciale del grafo del labirinto di Pac-Man (in alto). I vertici sono ordinati da in alto a sinistra a in basso a destra. Rappresentazione vertici-archi per lo stesso grafo (in basso)

Pac-Man esplora il grafo scegliendo una direzione per ogni vertice che incontra. Diverse sequenze di gioco produrranno quindi decisioni di direzione diverse e quindi grafi diversi (vedi figura 9).

Comunque, se Pac-Man ha completato il labirinto, ci aspettiamo che ci sia una relazione specifica tra i grafi prodotti.

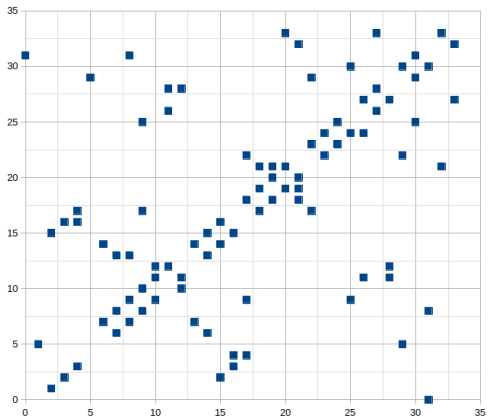
Diversamente, se Pac-Man viene mangiato da un fantasma

prima di completare il gioco, ovviamente non esplorerà l'intero labirinto, il che significa che ogni corsa produrrà casualmente un grafo diverso.
Esistono quindi due diverse situazioni:

a. i grafi prodotti corrispondono all'intero labirinto
b. i grafi prodotti non corrispondono all'intero labirinto

in entrambe le situazioni non si prevede che il grafo prodotto sia lo stesso

Ovviamente si può immaginare che, dato un labirinto, il suo grafo non possa dipendere fortemente dal percorso specificato scelto per esplorarlo. Infatti, se due grafi, nonostante il loro diverso aspetto, corrispondono allo stesso labirinto, si dice che sono isomorfi, cioè è possibile ottenere l'uno dall'altro con una semplice permutazione delle etichette dei loro vertici.
Questo è il caso di due grafi ottenuti da due esecuzioni di Pac-Man completate (tutti gli archi attraversati). D'altra parte, i grafi ottenuti dalle partite vinte dai fantasmi saranno generalmente incompleti (vedi fig. 10) e l'isomorfismo tra loro non è garantito. Tuttavia, questi grafi incompleti sono subisomorfi a quelli completi.



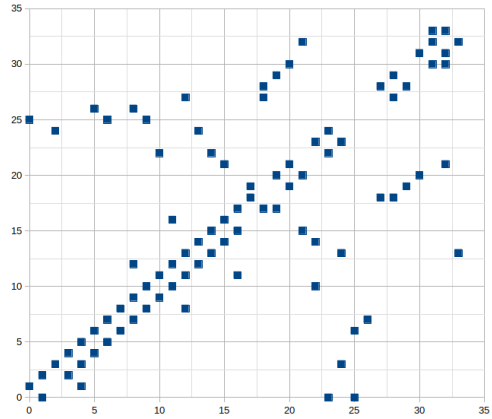


Fig. 9 - Rappresentazione matriciale del grafo del labirinto di Pac-Man. I vertici vengono ordinati non appena vengono scoperti dall'agent Pac-Man che vaga per il labirinto. Le immagini in alto e in basso corrispondono a due diverse serie in cui Pac-Man ha completato l'intero labirinto.

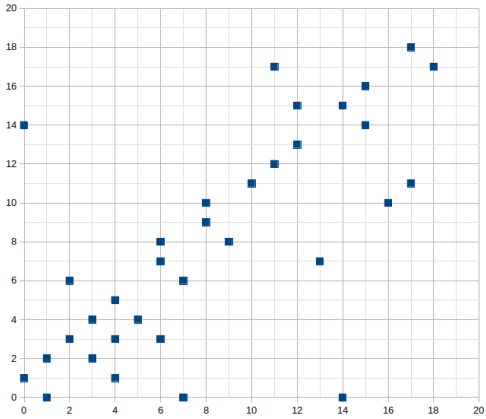


Fig. 10 - Rappresentazione matriciale del grafo a labirinto Pac-Man. Pac-Man non ha completato l'esplorazione del grafico.

Stampa del grafo

La rappresentazione *visiva* del grafo è utile per analizzare più facilmente alcune proprietà del grafo come il grado (valenza)

dei vertici. Questa rappresentazione può essere ottenuta tracciando su un piano cartesiano, i punti dati dalle coppie di indici dei vertici di ciascun arco.
Le figure 8, 9 e 10 sono state ottenute tracciando un set di dati prodotto durante l'esecuzione del codice `gioca_tuki_generagrafo.c` . Per testarlo, si decommentino le istruzioni:

```
//stampa(g);
```

poi si compili il progetto nuovamente (come mostrato sopra) e lo si esegua. Questo genererà il file `grafi.csv` nella directory in esecuzione. Se si vuol essere sicuri di generare il grafo completo, si avvii il gioco senza fantasmi:

```
~/grafi\$ ./tuki5.game 0
```

Analisi del grafo

Il file `grafi.csv` può essere usato per implementare facilmente il grafo in un programma basato sulla nota libreria C *igraph* .

Supponiamo che il contenuto del file sia il seguente:

```
 23,32,
 22,31,
 16,19,
 ...
 2,3,
 1,2,
 0,1,
```

può essere usato per inizializzare l'array `igraph_real_t edges1[]`nel codice seguente:

```
#include <igraph.h>
#include <stdio.h>
//gcc -o cmp confronta.c -I/usr/local/include/igraph -L/usr/

int main(void) {
  igraph_t graph1,graph2;
```

```
igraph_vector_t v1,v2;
igraph_vector_t result;

/* Tuki */
igraph_real_t edges1[] =
  {
     23,32, 22,31, 16,19, 19,5,
     20,19, 7,13, 18,9, 33,28,
     33,30, 28,33, 20,16, 21,20,
     5,21, 0,7, 8,6, 10,8, 14,
     21, 13,12, 9,13, 24,18, 32,
     25, 33,26, 30,33, 28,30,
     29,28, 10,29, 11,10, 12,11,
     15,12, 16,15, 9,18, 13,
     9, 22,8, 31,22, 31,23, 32,
     31, 32,23, 31,32, 23,31,
     24,23, 25,24, 26,25, 27,26,
     28,27, 30,28, 29,30, 28,
     29, 27,28, 26,27, 25,26, 24,
     25, 23,24, 22,23, 8,22,
     21,14, 20,21, 19,20, 5,19, 2,
     1, 6,2, 15,14, 12,15,
     17,11, 18,17, 17,18, 11,17, 16,
     10, 15,16, 14,15, 1,
     0, 14,0, 0,14, 7,0, 13,7, 12,
     13, 11,12, 10,11, 8,10,
     9,8, 8,9, 6,8, 7,6, 6,7, 4,1,
     6,3, 2,6, 3,2, 4,3, 5,
     4, 4,5, 3,4, 2,3, 1,2, 0,1
  };

/* PAC-MAN GRAPH */
igraph_real_t edges2[] =
  {
    0,2,0,3,0,5,1,6,1,8,1,9,2,0,
    2,3,2,10,3,0,3,2,3,4,
    3,10,4,3,4,5,4,12,5,0,5,4,5,6,
    6,1,6,5,6,7,7,6,7,8,7,13,8,1,
    8,7,8,9,8,11,9,1,9,8,9,11,10,2,
    10,3,10,14,11,8,11,9,11,17,
    12,4,12,13,12,15,13,7,13,12,13,
    16,14,10,14,15,14,17,14,20,
    15,14,15,12,15,18,16,13,16,17,
    16,19,17,11,17,14,17,16,17,
    23,18,15,18,19,18,21,19,16,19,
    18,19,22,20,14,20,21,20,24,
    20,30,21,18,21,20,21,26,22,19,
    22,23,22,27,23,17,23,22,23,
    29,23,31,24,20,24,25,24,30,25,
    24,25,26,25,32,26,21,26,25,
    26,27,27,22,27,26,27,28,28,27,
    28,29,28,33,29,23,29,28,29,
    31,30,20,30,24,30,32,31,23,31,
    29,31,33,32,30,32,25,32,33,
    33,28,33,31,33,32
```

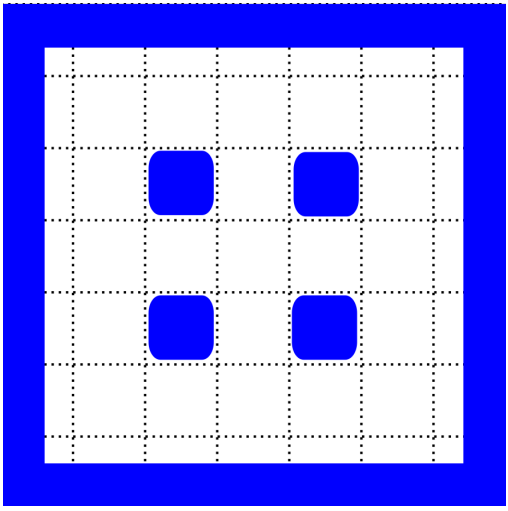
```
    };

    igraph_vector_view(&v1, edges1,
     sizeof(edges1)/sizeof(double));
    igraph_create(&graph1, &v1, 0,
     IGRAPH_UNDIRECTED);

    igraph_vector_view(&v2, edges2,
     sizeof(edges2)/sizeof(double));
    igraph_create(&graph2, &v2, 0,
     IGRAPH_UNDIRECTED);

    igraph_bool_t iso;
    igraph_subisomorphic(&graph2, &graph1,&iso);

    printf("%d\n",iso);
    igraph_destroy(&graph1);
    igraph_destroy(&graph2);

    return 0;
}
```

Si può immaginare che mentre il primo array rappresenta il grafo esplorato dall'agent, il secondo corrisponde a quello di figura 8.
La chiamata alla funzione `igraph_subisomorphic` verifica se `graph1` sia isomorfo a un sotto-grafo di `graph2` .
La chiamata a `igraph_isomorphic` (invece di `igraph_subisomorphic`) potrebbe restituire 0 poiché pacman non avrebbe potuto attraversare alcuni degli archi dove non ci sono pillole, ovvero 12-13, 13-16 , 16-19. 19-18, 18-15, 15-12.

In questa sezione abbiamo mostrato il metodo per creare un grafo archi-vertici anche in condizioni PO. Nella prossima illustreremo come usare questo grafo per ottimizzare l'agent di Pac-Man.

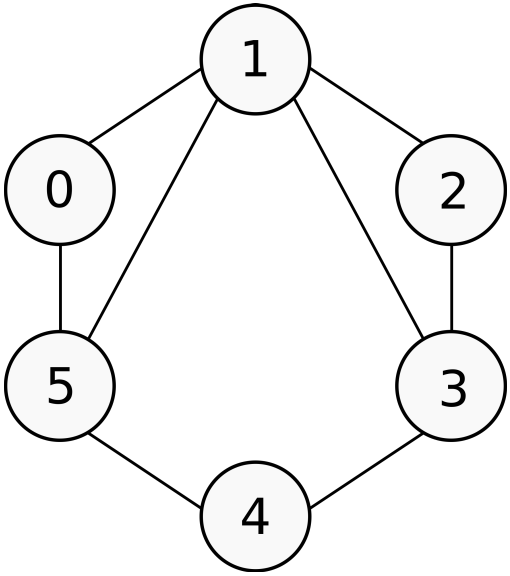
Box domande n.3

1. Nell'algoritmo `generagrafo`, cosa differenzia una cella-vertice da una cella-arco?
    a. Esistono solo celle-vertice, mentre gli archi sono collegamenti virtuali tra vertici
    b. Una cella-vertice ha almeno 3 celle vicine accessibili, mentre una cella-arco ne ha solo 2
    c. Niente, ad ogni esecuzione l'agent etichetta le celle come celle-vertice o celle-arco in maniera casuale
4. Trova il grafo di questo labirinto utilizzando l'algoritmo `generagrafo`, di quanti vertici e quanti archi è composto?



    a. 5 vertici, 8 archi
    b. 4 vertici, 4 archi
    c. 9 vertici, 12 archi

4. Trova il grafo dello stesso labirinto utilizzando l'algoritmo `boustrophedon`, di quanti vertici e quanti archi è composto?
    a. 9 vertici, 12 archi
    b. 25 vertici, 40 archi
    c. 21 vertici, 24 archi
4. Dato il seguente grafo, qual è la sua matrice corrispondente?



a.

0	1	0	0	0	1
1	0	1	1	0	1
0	1	0	1	0	0
0	1	1	0	1	0
0	0	0	1	0	1
1	1	0	0	1	0

0	1	0	0	0	1
1	0	1	0	0	0
0	1	0	1	0	0
0	0	1	0	1	0
0	0	0	1	0	1
1	0	0	0	1	0

b.

0	1	0	1	0	1
1	0	1	0	0	0
1	1	0	1	0	1
0	1	1	0	1	0
0	0	0	1	1	1
1	0	0	0	1	0

c.

Sfruttare il grafo

Ora che abbiamo visto come creare un grafo in tempo reale durante il gioco, possiamo chiederci come sfruttarlo. Dall'algoritmo boustrophedon abbiamo già imparato come garantire una copertura completa del labirinto. Ora vediamo come usare il grafo generato da `gioca_tuki_generagrafo.c` per trovare un percorso per raggiungere un vertice tra quelli già esplorati.

Si immagini di aver già esplorato una parte del labirinto e di voler tornare a un punto specifico (cella bersaglio). Il boustrophedon ci ha insegnato che dobbiamo camminare all'indietro fino a quando non ci imbattiamo nella cella bersaglio. Questo però significa anche sprecare molti turni di gioco e aumentare la probabilità che l'agent venga catturato dal fantasma. C'è un modo più efficiente per tornare indietro?
Ovviamente si possono calcolare tutti i possibili percorsi seguendo gli archi e i vertici fino a raggiungere la cella bersaglio e memorizzare la lunghezza di ciascuno. Al termine si può selezionare il percorso più corto.
Sfortunatamente questo metodo richiede un numero enorme di passaggi di calcolo per trovare la soluzione ottimale perché deve testare tutte le possibili combinazioni di vertici collegati tra loro.
Il numero di passaggi computazionali richiesti da questo algoritmo aumenta esponenzialmente con l'input, in questo caso, il numero di vertici e archi.
Ciò significa che questo algoritmo è adatto a grafi molto piccoli, ma se i vertici iniziano a crescere, impiega sempre più tempo.

La cattiva notizia è che non esistono tecniche matematiche per ridurre effettivamente la complessità di questo metodo. La buona notizia è che possiamo usare una funzione euristica per aiutarci.

Algoritmo A*

La nostra implementazione dell'algoritmo Astar è riportata nel codice seguente ed è ovviamente presentata nella libreria `libagri`.
L'algoritmo Astar (A*) sfrutta due concetti diversi: la distanza misurata e la distanza stimata tra due vertici. Per trovare il percorso da un vertice iniziale a quello bersaglio, l'algoritmo A* utilizza due funzioni: una denominata *esatta* e l'altra detta *euristica*.

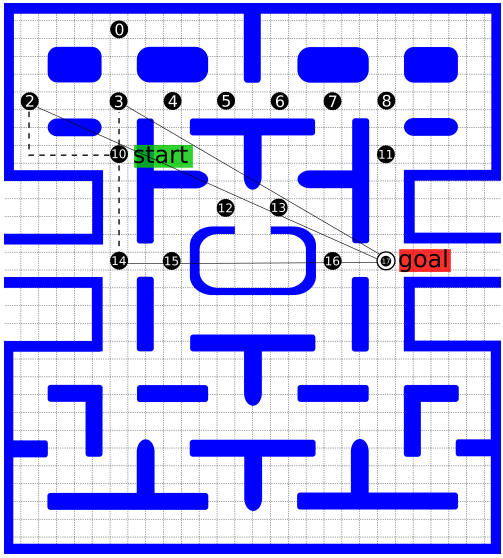


Fig. 11 - rappresentazione di A* per la ricerca del percorso dal vertice 10 al 17.

L'idea di base dell'algoritmo è quella di evitare di valutare tutti i possibili percorsi e concentrarsi principalmente su quelli più promettenti, ovvero i percorsi che *sembrano* i più brevi.

Per spiegare l'algoritmo diamo un'occhiata alla figura sopra. Pac-Man deve passare dal vertice 10 al vertice 17. Il primo passo di A* è creare una coda prioritaria e riempirla con i vertici vicini dell'attuale vertice, che alla prima iterazione è 10.

Una coda prioritaria è una specifica struttura dati, per esempio una lista collegata o uno stack, per memorizzare degli elementi ai quali è associata una etichetta di priorità.

La priorità nella coda dipende dalla distanza stimata dei vertici dall'obiettivo (cioè il vertice 17).
La distanza stimata di ciascun vicino si ottiene sommando la distanza esatta dal vertice corrente alla distanza euristica, che in questo caso corrisponde alla distanza cartesiana dal vertice obiettivo.
La distanza esatta è il numero di celle che separano un vertice dall'altro, ad esempio la distanza tra i vertici 10 e 2 è 7, la distanza tra i vertici 10 e 3 è 2 e la distanza tra i vertici 10 e 14 è 5. Si presti attenzione che in questa specifica implementazione non stiamo considerando le celle occupate dai vertici stessi, nel calcolo della distanza esatta.
Le distanze cartesiane dei vertici 2, 3 e 14 dal vertice goal (17) sono rispettivamente 22, 17 e 15. Di conseguenza, le loro distanze stimate risultano 7+22=29 per il vertice 2, 2+17=19 per il 3 e 5+15=20 per il 14.
Il vertice 3 è il primo nella coda prioritaria, in quanto la sua distanza stimata è la minore, seguito da 14 e 2.
Una volta riempita la coda, A* ripeterà nuovamente l'iterazione selezionando il primo vertice nella coda prioritaria, ovvero il vertice 3, come vertice corrente.
Ora che 3 è il vertice corrente, la procedura viene ripetuta considerando i vicini del vertice 3 che sono 2, 0 e 4. Seguendo questo metodo, il percorso da 10 a 17 è dato dalla seguente sequenza: 3, 4, 5, 6, 7, 8, 11 e 17.

A* in C

Per capire l'algoritmo A* seguiamo cosa succede dopo una chiamata alla nostra specifica implementazione dell'algoritmo:

```
agri_Via p = agri_astar
(10, 17, grafo, &distanza, &euristica,34)
```

Il vertice di partenza 10 viene inserito nell'elenco di priorità `candidati` , come si può vedere nel commento A-1.
Da A-1 segue un ciclo `while` che termina quando non ci sono più candidati o quando l'obiettivo (vertice 17) è stato raggiunto. Ad ogni iterazione viene estratto (pop) il vertice in cima alla lista, ovvero il vertice che ha la priorità più alta, ovvero la distanza più breve dall'obiettivo. Alla prima iterazione nell'elenco è presente solo il vertice di partenza, quindi l'operazione pop lo estrarrà dalla coda, lasciandola vuota.
In A-2 viene controllato se il vertice corrente e quello obiettivo corrispondono perché in tale situazione le iterazioni devono arrestarsi e deve essere restituito il percorso. Torneremo su questo punto più tardi.
In A-3 vengono analizzati i vicini del vertice 10 e la distanza dal vertice 17 viene calcolata per ciascuno di essi utilizzando sia la distanza esatta che quella euristica. Si noti che la funzione `agri_astar` accetta il puntatore alle funzioni che si desidera implementare per calcolare le distanze esatta ed euristica.
Ora siamo al punto critico poiché l'algoritmo deve decidere quale dei vertici vicini aggiungere alla coda di priorità. Al punto A-4 sono stati inizializzati due array: `gscore` che contiene la distanza esatta (misurata in unità di cella) di ciascun vertice dal vertice iniziale e `fscore` che è `gscore` sommato alla distanza euristica stimata del vertice dall'obiettivo.

Al momento dell'inizializzazione ogni elemento dei due array è impostato su infinito (9999) ad eccezione degli elementi corrispondenti al vertice iniziale che sono impostati rispettivamente su 0 e sulla distanza cartesiana.
Quindi, per il vertice iniziale, avremo `gscore [10] = 0` e `fscore [10] = 16` . In A-5 tutti i vertici adiacenti a quello iniziale soddisfano la `if-clause` perché il loro gscore è infinito e tutti saranno inseriti nella coda di priorità e il loro `gscore` verrà aggiornato con la loro distanza dal vertice iniziale, mentre il loro `fscore` verrà impostato sul loro `gscore` sommato alla distanza cartesiana dal vertice obiettivo.
I vertici 2, 3 e 14 vengono quindi inseriti nella coda di priorità con il 3 in cima alla coda. Per ognuno dei vertici appena inseriti in coda, viene memorizzato, al rispettivo indice dell'array chiamato `precedente`, l'indice del vertice corrente, ovvero il 10. In pratica l'array `precedente` serve per ricostruire la sequenza dei vertici a ritroso una volta che si è raggiunto il vertice obiettivo.
Alla successiva iterazione, 10 non è più il vertice corrente, ma lo diventa il vertice 3 a causa dell'operazione pop al punto A-6. Il blocco `if` nel punto A-2 non viene eseguito perché il vertice dell'obiettivo è 17 ed è diverso da 3. Il codice salta quindi in A-3 dove sono considerati i vicini del vertice 3, che sono: 2, 0, 4 e 10.
A questo punto siamo di nuovo in A-5 dove esiste la possibilità che il `gscore` del vertice 2 possa essere aggiornato. In effetti è possibile che il percorso che porta al vertice 2 passando per il 3 sia più corto di quello che lo connette direttamente al 10.

Questo pezzo di codice è molto importante nei grafi in cui la lunghezza degli archi è molto diversa dalla distanza cartesiana tra i vertici, comunque non gioca un ruolo importante nel labirinto di

L'algoritmo procede in questo modo fino a quando il vertice 17 risulta essere uno dei vicini del vertice corrente e viene quindi creato il percorso completo.
Alla fine la `if-clause` nel punto A-2 è vera e il blocco di codice viene eseguito. Nel loop A-7 tutti i vertici del percorso vengono scorsi all'indietro assegnando `corrente = precedente [corrente]` .
L'elenco dei vertici viene memorizzato nello heap e un puntatore al primo byte dell'array `precedente` viene restituito al chiamante della funzione.

```
agri_Via agri_astar(int start, int goal,
                    agri_Vertex * agri_Vertices_Colligati,
                    double (*spatium)(int ab, int ad),
                    double (*euristica)(int ab, int ad),
                    int nmembri
                    )
{
  if(start==goal)
    {
      return 0;
    }
  double fscore[nmembri];
  double gscore[nmembri];
  int precedente[nmembri];

  //Nodi in valutazione per il path da start a goal
  Ordo candidati = 0;

  /* A-4 */
  for(int i=0;i<nmembri;i++)
    {
      fscore[i] = INFINITO;
      gscore[i] = INFINITO;
    }
  gscore[start] = 0;
  fscore[start]=euristica(start,goal);

  Ordo_insero_nodus
  (&candidati,start,(1./fscore[start]));

  /* A-1 */
  while(candidati != 0)
    {
      fflush(stdout);
      /* A-6 */
```

```
int corrente = Ordo_pop(&candidati);

/* A-2 */
/* Arrivato al nodus goal torna il cammino */
if(corrente == goal)
  {

    fflush(stdout);
    int i = 0;
    int aux[nmembri];

    /* A-7 */
    do{
      aux[i]=corrente;
      corrente=precedente[corrente];
      i++;
    }while(corrente != start);

    agri_Via percorso = malloc((i+1)*sizeof(int));

    int j =0 ;
    for(int k = i-1;k >=0 ;k--,j++)
      {
        percorso[j] = aux[k];

      }
    // Segnale di fine percorso
    percorso[j]=-1;
    return percorso;
  }

/* A-3 */
/* Nel labirinto di Pac-Man ci sono al massimo
4 vicini */
int vicino[PORTE];
for(int i=0; i<PORTE; i++)
  {
    vicino[i] =
    agri_Vertices_Colligati[corrente].ianua[i];

  }

for(int i =0; i<PORTE; i++)
  {
    int iv = vicino[i];

    /* ogni nodus è predisposto per 4 vicini,
    ma molti nodi ne hanno solo 3*/
    if(iv == -1)continue;

    double d =  spatium(corrente, iv);
    double tent_gscore = gscore[corrente]+d;

    /* A-5 */
    if(tent_gscore <= gscore[iv])
```

```
            {
              /* Trovato cammino migliore a questo nodus
              passando per corrente*/
              precedente[iv] = corrente;
              gscore[iv] = tent_gscore;
              fscore[iv] = gscore[iv]+euristica(iv, goal);

              Ordo_insero_nodus
              (&candidati,iv, ( 1./fscore[iv]));

              fflush(stdout);
            }
        }
    }
  return 0;
}
```

Navigazione nel grafo esplorato usando l'algoritmo A *

L'algoritmo A * combinato con la rappresentazione a grafo del labirinto Pac-Man offre l'opportunità di creare un agent in grado di muoversi in modo deterministico all'interno del labirinto.

L'idea è quella di codificare un agent che esplori il labirinto per creare il grafo e nel frattempo usi il grafo creato per guidare se stesso nell'esplorazione.

Specifiche dell'agent

Chiediamo semplicemente all'agent di tornare periodicamente a un certo punto del labirinto.
Gli chiediamo di tornare a un dato vertice dopo aver aggiunto un certo numero di vertici al suo grafo. In altre parole, l'agent deve: tornare a casa per pranzo dopo una giornata di lavoro!
La scelta di innescare il ritorno dell'agent in base al numero di vertici aggiunti è solo a titolo di esempio di possibili usi dell'A*. Si potrebbero fare diverse scelte come parametro di innesco del ritorno, come ad esempio il tempo di

vagabondaggio o il numero di passi (turni di gioco) eseguiti.
Ci chiediamo però cosa succede se Pac-Man incontra un fantasma. In questo caso, chiediamo all'agent di "sospendere" la sua esplorazione o di sospendere il suo ritorno a casa e fuggire dal fantasma fino a quando il fantasma non è più nelle vicinanze. Queste due semplici regole saranno la logica che guiderà il nuovo agent.

Implementazione

Per implementare la specifica dell'agent, sono necessari i seguenti passaggi:

1. creare il grafo durante il vagabondaggio di Pac-Man nel labirinto
2. contare i nuovi vertici aggiunti al grafo
3. chiamare l'algoritmo A* quando è stato raggiunto il numero di nuovi vertici aggiunti al grafo
4. memorizzare il percorso in memoria come un array di vertici
5. trovare la direzione per il prossimo vertice nel percorso restituito

Passo 1

```
agri_Colligatio colligatio;
agri_Vertex v_a, v_da;
v_a = agri_Verticem_creo(vertice_a,i,j);
v_da = agri_Verticem_creo(vertice_da,i_da,j_da);

distanze[vertice_da][vertice_a] = longitudo_colligatio;

colligatio.ad = v_a;
colligatio.ab = v_da;
colligatio.longitudo = longitudo_colligatio;
colligatio.meta = direzione_arrivo;
colligatio.discessus = direzione_partenza;
agri_Colligationem_insero(&g, colligatio);
nodo_rilevato = true;
```

Passo 2

```
int vertice_a = agri_Verticem_quaero(g,i,j);
if(vertice_a<0)
  {
    vertice_a = vertici_contati;
          vertici_contati++;
          nodi_percorsi++;
        }
```

Passi 3-4

```
/* Il percorso */
static int * percorso_fuga = 0;
/* Copia del percorso */
static int * copia;
...
/* Chiama A* per calcolare il percorso */
percorso_fuga = agri_astar(...);
...
/* Libera lo heap */
if(copia) free(copia);
```

Passo 5

```
/*
  Cerca la direzione per raggungere il
  prossimo vertice
*/
if(vert_disp[vertice_da].ianua[SINISTRA] ==
  ndice_nodo)
            {
              ld = SINISTRA;
            }
if(vert_disp[vertice_da].ianua[DESTRA] ==
  indice_nodo)
            {
              ld = DESTRA;
            }
if(vert_disp[vertice_da].ianua[SU] ==
   indice_nodo)
            {
              ld = SU;
            }
if(vert_disp[vertice_da].ianua[GIU] ==
   indice_nodo)
            {
              ld = GIU;
            }
```

Il codice completo è presentato in appendice.

Compilazione ed esecuzione

Compilare ed eseguire come indicato nel seguito:

```
~/grafi\$ ./compila.sh
PacMan/gioca_tuki_esploraeritorna.c 1 50
```

Poi esegui specificando 1) zero fantasmi:

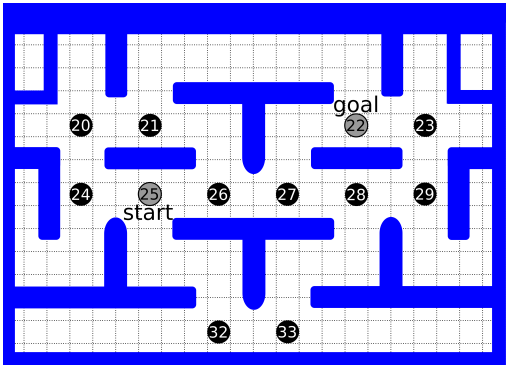
```
~/grafi\$ ./tuki5.game 0
```

esegui specificando 2) uno o più fantasmi:

```
~/grafi\$ ./tuki5.game 1
```

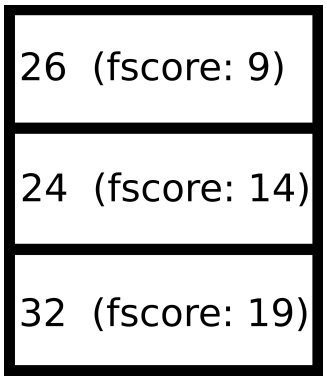
Box domande n.4

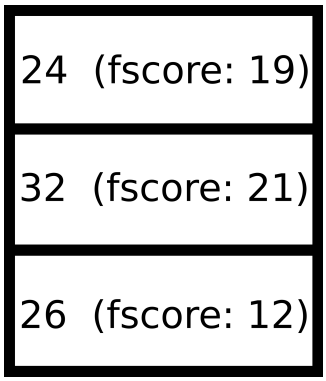
1. Il Pac-Man si trova sul vertice 25 e deve reggiungere il 22. L'agent che lo guida è l'`astar`. Quale di queste code di priorità è quella giusta?



a.

32 (fscore: 21)
24 (fscore: 19)
26 (fscore: 12)

b. 

c. 

4. Al termine dell'esecuzione dell'algoritmo astar, cosa contiene l'array `precedente`?
    a. La lista di tutti i vertici considerati durante la ricerca del percorso
    b. I vicini del vertice goal
    c. Il percorso sub-ottimale restituito dall'algoritmo per raggiungere il vertice goal partendo dal vertice start
4. Cosa contengono gli array `fscore` e `gscore`?
    a. gscore: distanza esatta di ogni vertice dal vertice start
       fscore: gscore + distanza cartesiana di ogni vertice dal vertice goal
    b. gscore: distanza esatta di ogni vertice dal suo precedente
       fscore: gscore + distanza esatta do ogni

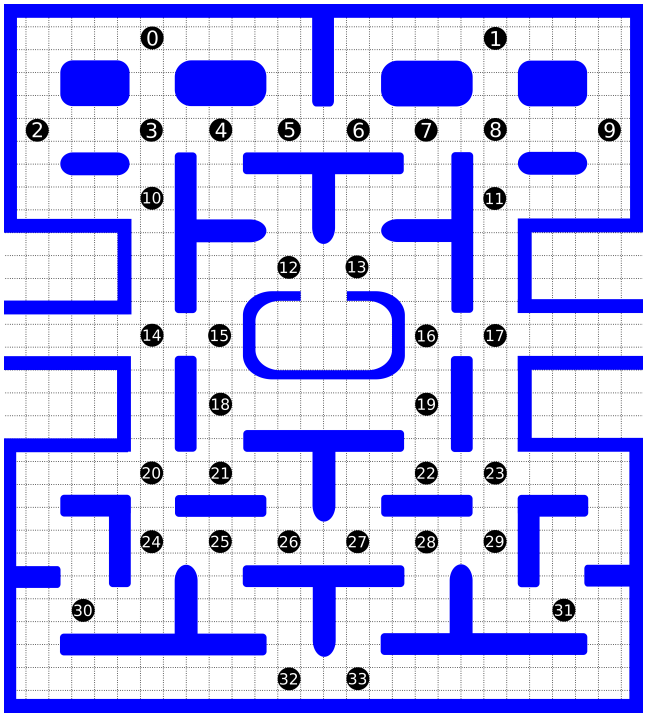
vertice dal vertice goal

c. gscore: distanza stimata di ogni vertice dal vertice start
   fscore: distanza cartesiana di ogni vertice dal vertice goal

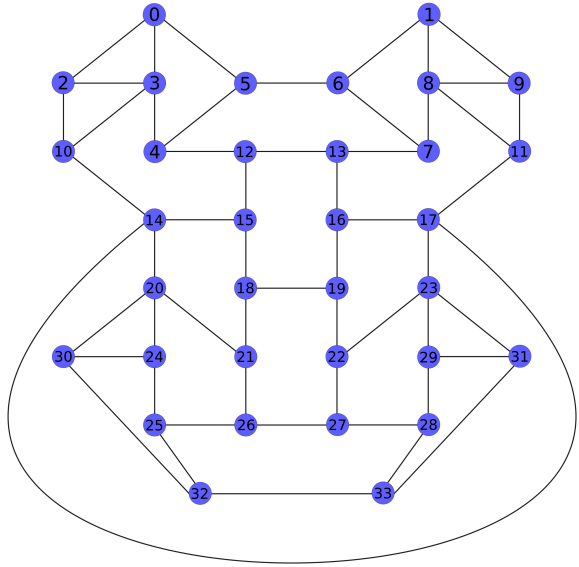
FULL OBSERVABILITY

In questa sezione cambiamo il paradigma del gioco e assumiamo la osservabilità totale. La piena (full) osservabilità (FO) significa che l'agent ha piena conoscenza della struttura del labirinto nel momento in cui inizia il gioco. Ciò consente all'agent di scegliere una sequenza di vertici per ridurre al minimo la lunghezza del percorso.

È facile intuire che il problema di ridurre al minimo (minimizzare) la lunghezza del percorso ha molte ricadute pratiche nel mondo reale. Ad esempio è importante in tutte le applicazione di logistica, come ad esempio nella definizione del percorso degli addetti ai carrelli in un magazzino merci.



12.a



12.b

Fig. 12 - La figura mostra l'indice (o etichetta) di ogni incrocio nel labirinto (a) e le etichette vengono utilizzate per generare il grafo del labirinto(b)

Preparazione del grafo

Il labirinto di Pac-Man viene trasformato in un grafo dichiarando una matrice di vertici e inizializzando ciascuna porta con i vicini del vertice (i campi `ianua`, che in latino significa porta).

```
grafo[0].linea = 3;
grafo[0].columna = 6;
grafo[0].index = 0;
grafo[0].ianua[SINISTRA] = 34;
grafo[0].ianua[DESTRA] = 35;
grafo[0].ianua[SU] = -1;
grafo[0].ianua[GIU] = 3;
...
grafo[34].ianua[DESTRA] = 0;
...
```

```
grafo[35].ianua[SINISTRA] = 0;
...
grafo[3].ianua[SU] = 0;
```

Questo codice fa riferimenro al labirinto di figura 13 in cui sono stati aggiunti alcuni vertici supplementari.

Percorso sub ottimale per coprire il labirinto

Il percorso che Pac-Man deve seguire per coprire tutto il labirinto viene deciso "fuori dal gioco", cioè a design-time dal programmatore e l'agent ne viene "informato".

L'ideale sarebbe quello di percorrere ogni corridoio del labirinto una sola volta, in questo modo si ridurrebbe sicuramente al minimo il tempo di gioco. Nel linguaggio della teoria dei grafi, un percorso che non ripassi mai per uno stesso arco viene detto *percorso euleriano* dal matematico Eulero che per per primo studiò questo problema. La condizione necessaria per poter costruire un percorso euleriano è che il grado (cioè il numero di vicini) di ogni vertice sia un numero pari, condizione non presente nel labirinto del Pac-Man. Dovremo pertanto accontentarci di un percorso sub-ottimale.

Take home message: un percorso euleriano richiede un grafo che abbia tutti i vertici di grado pari.

La sequenza di vertici del percorso è memorizzata in un array di numeri interi che rappresentano l'indice di ogni vertice:

```
static int cam[]=
    {
      3,10,38,2,34,0,3,2,3,0,35,5,4,3,4,12,13,
      7,6,5,6,36,1,8,7,8,1,37,9,39,11,39,9,8,11,17,
```

```
    16,13,12,15,14,17,23,22,19,16,19,15,
    21,40,20,14,20,40,30,42,32,33,43,31,41,
    23,29,31,29,28,33,28,27,22,27,26,21,26,
    25,32,25,24,20,24,30
};
```

Questa sequenza di vertici è stata costruita in modo che Pac-Man, percorrendola, passi per tutti i corridoi.

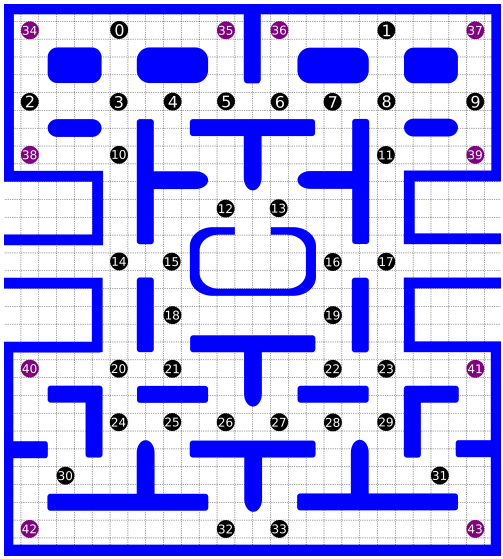


Fig. 13 - I vertici dal 34 al 43 sono stati aggiunti alla rappresentazione originale del grafo.

Ai 34 vertici di Pac-Man sono stati aggiunti altri dieci, che hanno lo scopo di forzare l'agent ad entrare in alcuni archi che altrimenti avrebbe evitato perché il percorso che li comprende non è il più breve e quindi non è contemplato dall'A*. Ad esempio, l'agent eviterebbe l'angolo in alto a sinistra passando da 2 a 0, preferendo passare dal vertice 3.
Quando il gioco inizia il vertice di destinazione è il 3 (vedi array `cam`) e il suo percorso dalla posizione iniziale di Pac-Man viene stabilito chiamando l'algoritmo A*.
Va notato che è possibile che la posizione iniziale di Pac-Man non sia un vertice, quindi deve vagare nel labirinto fino a raggiungere il primo vertice.

L'agent ha una variabile indice per navigare la matrice cam della sequenza di vertici da seguire per completare il percorso:

```
int indice_array_vertice_bersaglio = 0;
```

Tale indice non viene aggiornato fino a quando non viene raggiunto il vertice di destinazione, a quel punto viene incrementato di un'unità:

```
if(vertice_corrente ==
cammino[indice_array_vertice_bersaglio])
  {
    indice_array_vertice_bersaglio++;
    modo_gioco = ESPLORA;
    free(copia);
    copia = 0;
   ...
  }
```

Take home message: È possibile utilizzare una sequenza di vertici per specificare un percorso per un agent senza specificare tutte le celle, ad una ad una, per cui l'agent deve passare.

Il codice completo gioca_tuki_percorso.c si trova in appendice.

Compila ed esegui il codice

Compilare il codice con le seguenti opzioni:

```
~/grafi\$ ./compila.sh PacMan/gioca_tuki_percorso.c
1 50
```

Quindi esegui specificando 1) nessun fantasma:

```
~/grafi\$ ./tuki5.game 0
```

esegui specificando 2) uno o più fantasmi:

```
~/grafi\$ ./tuki5.game 1
```

Box domande n.5

1. Come deve essere un grafo perché sia possibile costruire un *percorso euleriano*?
    a. Tutti i vertici devono avere un numero pari di vicini
    b. Tutti i vertici devono essere di grado dispari
    c. Tutti i vertici devono avere almeno due vicini
4. Dato il seguente labirinto, quale tra questi percorsi permette di coprirlo interamente?
    a. 0-1-6-5-4-0-4-12-4-5-6-7-2-3-9-13 -9-8-7-8-11-10-5-10-14-10-11-15
    b. 0-4-12-4-5-6-1-6-7-2-3-9-13-9-8-11 -15-11-10-14
    c. 15-11-10-14-10-5-4-12-4-0-1-6-7-8-9-3-2
4. Quale tra questi pezzi di codice si riferisce a questo labirinto?
    a. grafo[5].linea = 4
       grafo[5].columna = 4
       grafo[5].index = 5
       grafo[5].ianua[SINISTRA] = 4
       grafo[5].ianua[DESTRA] = 6
       grafo[5].ianua[SU] = -1
       grafo[5].ianua[GIU] = 10

    b. grafo[6].linea = 4
       grafo[6].columna = 7
       grafo[6].index = 6
       grafo[6].ianua[SINISTRA] = 5
       grafo[6].ianua[DESTRA] = 7

grafo[6].ianua[SU] = -1
grafo[6].ianua[GIU] = 10

c. grafo[10].linea = 7
grafo[10].columna = 10
grafo[10].index = 10
grafo[10].ianua[SINISTRA] = 14
grafo[10].ianua[DESTRA] = 11
grafo[10].ianua[SU] = -1
grafo[10].ianua[GIU] = -1

Implementare la fuga nella funzione euristica

Fin'ora abbiamo ottenuto due risultati importanti. Il primo è che l'agent di Pac-Man riesce a percorrere l'intero labirinto senza il bisogno di essere guidato passo per passo, ma disponendo solo di una lista pre elaborata dei nodi da raggiungere in sequenza. L'agent sfrutta l'algoritmo A* per raggiungerli dalla posizione in cui si trova. Inoltre, quando Pac-man si sposta dalla sequenza di nodi per sfuggire agli attacchi dei fantasmi, l'algoritmo A* gli permette di recuperare rapidamente l'ultima posizione lasciata per fuggire e riprendere la sequenza di nodi programmata in precedenza.

Il secondo risultato è che Pac-Man riesce a completare il labirinto anche in presenza del fantasma Blinky, di cui l'agent è capace di evitare tutti gli attacchi. Questo secondo risultato però non ci deve entusiasmare troppo, infatti dato che ogni vertice ha almeno tre vertici direttamente connessi, cioè nel linguaggio matematico il grafo ha *grado* minimo pari a tre, è normale che l'agent riesca sempre ad evitare Blinky, visto che i due hanno la stessa velocità.

Il meccanismo con il quale l'agent riesce a sfuggire all'attacco di Blinky è basato solo sulla sua capacità di *vedere* il fantasma quando questi si trova ad una certa distanza e quindi di invertire la propria direzione per evitare la *collisione*. In questo paragrafo vediamo invece come sfruttare l'algoritmo A* per evitare Blinky senza bisogno di sfuggirgli direttamente.

Tool di test per A*

Per meglio comprendere la logica che useremo di seguito conviene scrivere un tool per testare i percorsi calcolati dalla funzione A* senza bisogno di lanciare il gioco. Il codice del tool è piuttosto semplice, si tratta di scrivere una funzione `main` che accetti come argomento il nodo di partenza e il nodo di destinazione per i quali si vuole calcolare un percorso nel labirinto di Pac-Man. La funzione `main` deve quindi chiamare la funzione A* passandogli come argomento anche le due funzioni per la stima e il calcolo esatto della distanza. Ovviamente queste due funzioni devono essere definite nel codice del tool e le copiamo esattamente dall'ultima implementazione che abbiamo dato dell'agent. In effetti si tratta di un codice molto semplice, ma per completezza anche questo è riportato in appendice. Si consiglia di editare il codice e di compilarlo. Nel seguito faremo riferimento all'eseguibile chiamandolo `test_astar`.

Un ruolo nuovo per l'euristica

Si è visto che la funzione euristica può essere usata per stimare il percorso più breve tra il nodo di partenza ed il nodo di arrivo.
Nell'esempio analizzato abbiamo implementato tale funzione come il calcolo della distanza cartesiana tra le celle relative ai due nodi. È importante capire che la scelta dell'implementazione della funzione euristica è totalmente arbitraria e quindi possiamo implementarla a piacere. È chiaro però che la sua implementazione avrà effetto sul calcolo del percorso ma, come vedremo ora, questo è esattamente quello che vogliamo ottenere.

Usiamo il tool appena scritto (che si trova anche su GitHub) per analizzare il percorso calcolato da A* dal vertice di

partenza 26 al vertice di arrivo 3, che sono i vertici iniziali del percorso sub-ottimale definito nel paragrafo precedente.

```
~/grafi\$ ./test_astar 26 3
```

L'output ci fornisce un percorso costituito dalla lista di vertici: {21,18,15,14,10,3}

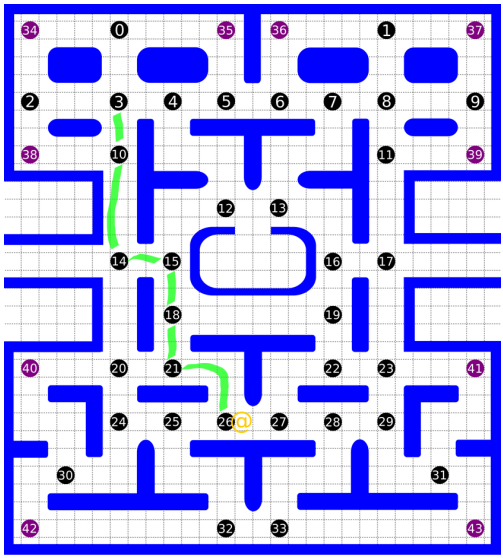


Fig. 14 - Percorso calcolato da A* tra i vertici 26 e 3 è tracciato in verde. Pac-Man è rappresentato con il simbolo @ vicino al vertice 26.

Vogliamo provare ad evitare che il percorso di Pac-Man passi per il vertice 15. Per farlo proveremo a modificare la distanza stimata del vertice 15 rispetto al 26, in questo modo l'algoritmo A* preferirà selezionare un percorso che non comprenda il vertice 15.
Modifichiamo la funzione `euristica_h` aggiungendo la seguente condizione prima del `return`:

```
if(start == 15)
    {
      return 200;
    }
```

La funzione è quindi definita come segue:

```
double euristica_h(int start, int goal)
{
  int x1,x2,y1,y2;
  double d;
  x1 = grafo[start].columna;
  y1 = grafo[start].linea;
  x2 = grafo[goal].columna;
  y2 = grafo[goal].linea;
  //Euristica uguale al quadrato della distanza euclidea
  d = (x1-x2)*(x1-x2)+(y1-y2)*(y1-y2);
  if(start == 15)
    {
      return 200;
    }
  return sqrt(d);
}
```

Il senso della modifica risulta chiaro se si ricorda che la funzione euristica viene chiamata dalla funzione A* per ogni vertice che deve essere valutato al fine della definizione del percorso. Ad ogni chiamata l'indice del vertice in valutazione è memorizzato nel parametro `start` mentre nel parametro `goal` è fisso l'indice 26 (con riferimento all'esempio che stiamo valutando).
La condizione:

```
if(start == 15)
    {
      return 200;
    }
```

fa si che il vertice 15 risulti molto lontano dal 26, finendo per essere scartato dal percorso calcoalto dall'algoritmo.

Si compili il codice con la modifica apportata e si esegua ancora con la stessa linea di comando:

```
~/grafi\$ ./test_astar 26 3
```

L'output ci fornisce un percorso diverso da prima, costituito dalla lista di vertici: {25,24,20,14,10,3}

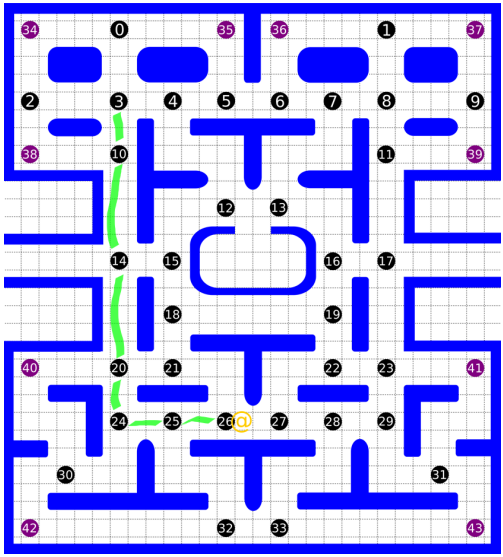


Fig. 15 - Percorso calcolato da A* tra i vertici 26 e 3. Pac-Man è rappresentato con il simbolo AT vicino al vertice 26.

La funzione euristica, così come la distanza esatta non devono rispettare delle leggi imposte dall'esterno, ma possono essere implementate a piacere per applicare la logica del problema a cui si stanno applicando i grafi.
Per esempio i navigatori satellitari, quando calcolano un percorso da seguire con l'automobile, non sono costretti a calcolare per forza il percorso più breve ma possono evitare dei vertici (per esempio strade o incroci) in cui il traffico sia in quel momento troppo intenso.
In questo primo semplice esempio abbiamo visto come evitare un vertice in modo `hard coded` cioè il vertice da evitare è stato memorizzato già nel codice prima di compilarlo e di eseguirlo. Nel prossimo paragrafo vediamo come stabilire un vertice da evitare a *runtime* in base alla configurazione istantanea del gioco.

Test dinamico Pac-Man e Blinky

Vediamo ora come mettere a frutto l'idea sviluppata nel paragrafo precedente applicandola al contesto di fuga di Pac-Man dal singolo fantasma Blinky.

Per semplificare la trattazione eseguiamo una serie di *esperimenti* usando il tool di test, e riserviamo l'implementazione nell'agent di Pac-Man a quando ci saremo chiariti bene le idee su come procedere.

Faremo anzitutto le seguenti ipotesi semplificative:

1. Pac-Man si muove da vertice a vertice
2. Blinky segue un percorso predefinito

che ci permettono di analizzare il problema in modo prototipale sfruttando il tool di test. Rispetto a quanto implementato nel listato `gioca_tuki_percorso.c` mancherà lo stato di NAVIGA, cioè lo stato in cui si trova l'agent mentre si sposta da un vertice all'altro.

Come primo passo modifichiamo la funzione `main` in modo che percorra iterativamente tutti i vertici calcolati da A* fino al raggiungimento del `goal`.

```
/*_________________________________
 *
 * MAIN 2
 */
int main(int argc, char * argv[])
{
  int s = atoi(argv[1]);
  int g = atoi(argv[2]);

  collega_tuki_nodi();

  while(s != g)
    {
      printf("%d- \t",s);
      agri_Via pc =
        agri_astar
        (s,g,grafo,&distanza_esatta,&euristica_h,44);
      s = *pc;
      while(*pc>=0)
        {
```

```
            printf("%d\t",*pc);
            pc++;
          }
        printf("\n");
    }
}
```

In questo codice il percorso viene continuamente ricalcolato ad ogni vertice raggiunto. Questo comportamento è diverso da quello implementato in precedenza nel codice dell'agent, dove per *risparmiare* CPU, una volta ottenuta la lista dei vertici da percorrere non si richiamava più l'A* a meno che un attacco di Blinky non portasse Pac-Man fuori rotta.

Compilando ed eseguendo di nuovo il codice:

```
~/grafi\$ ./test_astar 26 3
```

si ottiene l'output seguente:

```
26-> 21 18 15 14 10 3
21-> 18 15 14 10 3
18-> 15 14 10 3
15-> 14 10 3
14-> 10 3
10-> 3
```

dove ogni riga rappresenta la lista di vertici da percorrere per arrivare al traguardo.

Ora inseriamo nella `main` una semplice simulazione del movimento di Blinky. In pratica creiamo un array che contiene la sequenza dei vertici del percorso del fantasma e, come già fatto per Pac-Man, supponiamo che anche il rosso antagonista si muova da vertice a vertice senza perdere tempo lungo gli archi.
Dichiariamo le variabili globali

```
int b_iter[]={13,12,15,14,17,14,15,12,13,16,17,14,10};
int passo;
```

in testa al listato e modifichiamo la `main` incrementando la variabile globale `passo` prima della chiusura del ciclo:

```
while(s != g)
   {
     ...
     printf(":%d\n",b_iter[passo]);
     passo += 1;
   }
```

Manca ancora una modifica da fare sulla funzione `euristica_h`, infatti ora vogliamo che la funzione restituisca una distanza *molto grande* nel caso in cui il vertice `start` sia un vicino del vertice su cui si trova Blinky. Il codice che implementa questa variazione è il seguente:

```
if(grafo[b_iter[passo]].ianua[SINISTRA] == start ||
    grafo[b_iter[passo]].ianua[DESTRA] == start ||
    grafo[b_iter[passo]].ianua[SU] == start ||
    grafo[b_iter[passo]].ianua[GIU] == start)
   {
     return 200;
   }
```

Risultati del test

Eseguendo:

```
~/grafi\$ ./test_astar 26 3
```

con gli stessi parametri usati prima viene prodotto l'output seguente:

```
26-> 21 18 15 14 10 3 :13
21-> 20 14 10 3 :12
20-> 24 25 26 27 28 29 23 17 11 8 7 6 5 4 3 :15
24-> 25 26 27 22 19 16 13 12 4 3 :14
25-> 24 20 14 10 3 :17
24-> 25 26 27 22 19 16 13 12 4 3 :14
25-> 26 27 28 29 23 17 11 8 7 6 5 4 3 :15
26-> 25 24 20 14 10 3 :12
25-> 24 20 14 10 3 :13
```

```
24-> 20 14 10 3 :16
20-> 14 10 3 :17
14-> 15 12 4 3 :14
15-> 14 10 3 :10
14-> 10 3 :2
10-> 3 :0
```

Ogni riga riporta la lista dei vertici su cui muovere Pac-Man a partire dall'ultima posizione occupata, per esempio la prima riga presuppone che il vertice di partenza sia il 26 e il primo spostamento sia verso il 21; allo stesso modo la seconda riga assume come vertice di partenza il 21 e il 20 come il vertice su cui muovere Pac-Man. Al termine di ogni riga, dopo i due punti, è stampata la posizione di Blinky.

Analizziamo i risultati del test e, per meglio visualizzarli, riportiamo il percorso sul labirinto anche se il test è stato condotto in un grafo più semplice un cui sostanzialmente i vertici sono collegati gli uni agli altri senza i corridoi presenti nel labirinto reale.

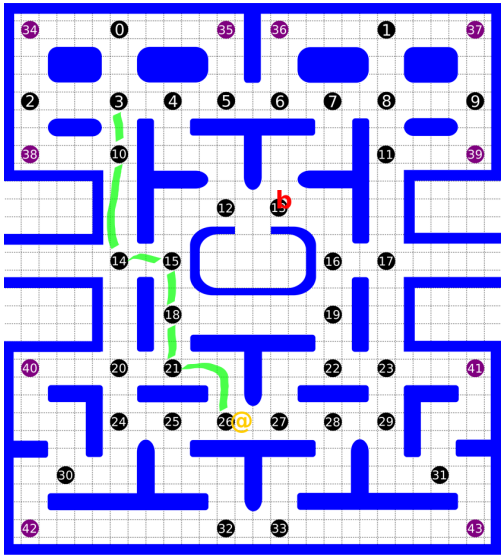


Fig. 16 - Percorso calcolato da A* tra i vertici 26 e

3. Pac-Man è rappresentato con il simbolo AT sul vertice 26 mentre Blinky è rappresentato con la lettera b sul vertice 13.

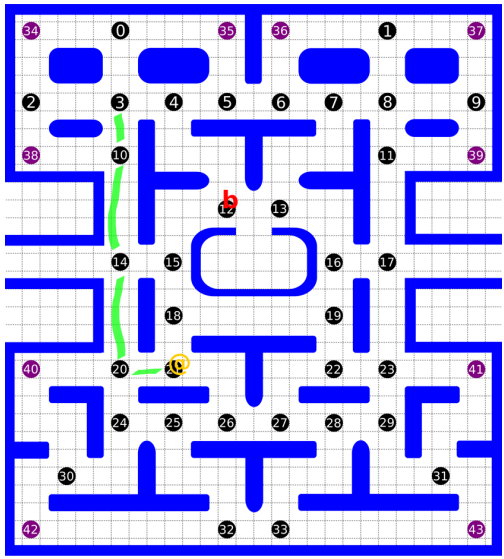


Fig. 17 - Percorso calcolato da A* tra i vertici 21 e 3. Il percorso risulta variato rispetto alla iterazione precedente che comprendeva il vertice 14 in quanto ora Blinky può raggiungere tale vertice in un solo movimento.

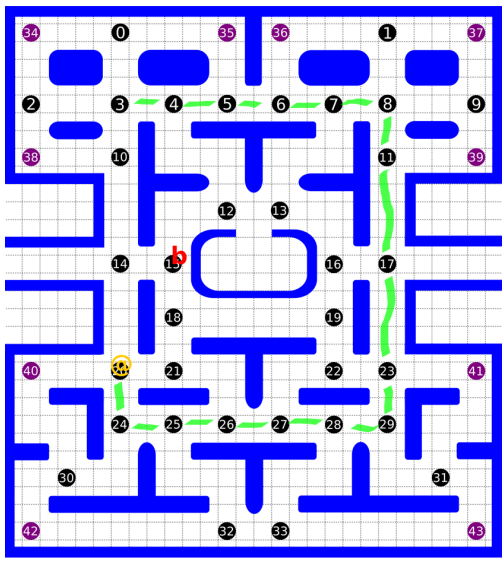


Fig. 18 - Percorso calcolato da A* tra i vertici 20 e 3. Il nuovo percorso risulta il più conveniente come compromesso tra brevità e distanza da Blinky.

Come si è detto, per ogni ciclo, oltre a riportare il cammino pianificato da A*, la `main` stampa a terminale il vertice su cui si trova Blinky. Inizialmente Blinky è in posizione 13 e l'A* calcola il percorso come visto in precedenza (figura 16). Quando però Pac-Man si sposta sul vertice 21, Blinky si muove sul 12 che ha come vicino il 15, quindi la seconda riga risulta modificata rispetto alla prima perché A* ha ricalcolato il percorso per evitare il nodo 15 (figura 17). Quando Pac-Man si muove sul nodo 20, Blinky lo anticipa sul 15, bloccandogli il cammino verso il 14 e costringendolo a fare marcia indietro verso il nodo 24 e ad intraprendere un cammino che circumnaviga la *casa dei fantasmi* che, per quanto risulti lungo, viene comunque valutato più conveniente rispetto a quello che passa per i nodi prossimi a Blinky (figura 18).
Il resto degli spostamenti può essere dedotto leggendo l'output qui sopra. Il risultato interessante che abbiamo ottenuto è di avere implementato un meccanismo di fuga che non è basato su un controllo diretto della prossimità del fantasma, ma che poggia direttamente su una pianificazione del percorso.

Il risultato che abbiamo ottenuto è molto incoraggiante perché indica una via di *navigazione ragionata* in cui le proprietà dei grafi possono essere usate per evitare anche ostacoli dinamici come appunto la presenza dei fantasmi.
Prima di procedere e vedere come riportare questi risultati direttamente dentro la piattaforma di gioco, continuiamo con i test e vediamo se è possibile anche implementare la stessa logica quando i fantasmi sono in numero maggiore di uno.

Test dinamico Pac-Man e Ghost Team

Iniziamo inserendo nel gioco il fantasma Pinky accanto a Blinky. In questo contesto i nomi ci servono solo a titolo esemplificativo perché i percorsi di attacco che stiamo usando per i fantasmi non rappresentano le loro personalità che sono state spiegate nei capitoli precedenti.

Per simulare la presenza di Pinky aggiungiamo l'array `p_iter` alle variabili globali, dichiarandolo subito dopo quello di Blinky.

```
int p_iter[]={12,13,16,19,22,23,17,11,8,7,6,5,4,3,2,0};
```

A questo punto è necessario modificare anche la funzione `euristica_h` in modo che tenga conto di entrambi i fantasmi. Per farlo aggiungiamo la condizione seguente subito sotto la precedente condizione già scritta che si riferisce a Blinky:

```
if(grafo[p_iter[passo]].ianua[SINISTRA] == start ||
     grafo[p_iter[passo]].ianua[DESTRA] == start ||
     grafo[p_iter[passo]].ianua[SU] == start ||
     grafo[p_iter[passo]].ianua[GIU] == start)
    {
      return 200;
    }
```

Modifichiamo la `main` in modo che venga stampata anche la posizione di Pinky oltre a quella di Blinky:

```
printf(":%d :%d\n",b_iter[passo],p_iter[passo]);
```

Eseguiamo di nuovo con lo stesso argomento:

```
~/grafi\$ ./test_astar 26 3

26-> 25 24 20 14 10 3 :13 :12
25-> 24 20 14 10 3 :12 :13
24-> 20 14 10 3 :15 :16
20-> 24 25 26 27 28 29 23 17 11 8 7 6 5 4 3 :14 :19
```

```
24-> 20 14 10 3 :17 :22
20-> 21 18 19 16 13 12 4 3 :14 :23
21-> 18 15 14 10 3 :15 :17
18-> 21 20 14 10 3 :12 :11
21-> 18 15 14 10 3 :15 :8
18-> 15 12 4 3 :14 :7
15-> 12 4 3 :20 :6
12-> 15 14 10 3 :21 :5
15-> 14 10 3 :18 :4
14-> 10 3 :15 :3
10-> 3 :14 :2
```

Analizzando le righe dell'output si vede che il calcolo del percorso procede in modo simile all'esempio precedente in cui era coinvolto un singolo fantasma.
La situazione però cambia decisamente alla decima riga quando il percorso calcolato dall'A* per Pac-Man si rivela insicuro in quanto passa accanto a Blinky. È naturale chiedersi perché sia stato calcolato un percorso così periocoloso. La risposta però diventa naturale guardando la figura 19. Infatti qualsiasi percorso si provi a considerare nel labirinto, per arrivare al vertice 3 partendo dal 18 è necessario passare in prossimità o del veritce 14 su cui è presente Blinky o del vertice 7 su cui è invece presente Pinky, quindi non esiste, al momento del calcolo, un percorso sicuro. Dal momento che la prossimità a Pinky é valutata nello stesso modo della prossimità a Blinky (cioè una distanza di 200) l'algoritmo premia il percorso complessivamente più breve, cioè quello che passa accanto a Blinky.

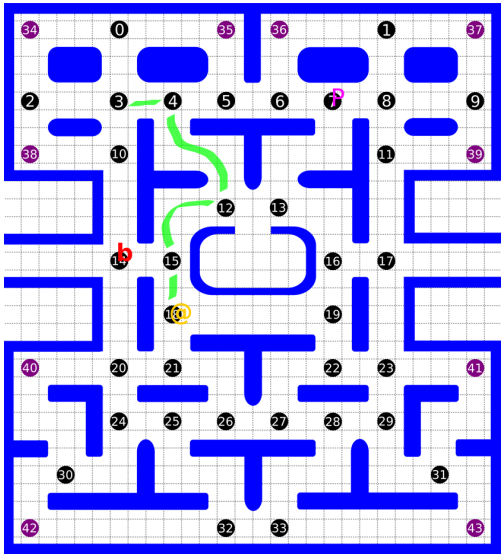


Fig. 19 - Percorso calcolato da A* tra i vertici 18 e 3. Il percorso comprende un veritce contiguo a quello occupato da Blinky.

Il ragionamento appena svolto ci spinge a verificare cosa succede modificando la risposta della `euristica_h` in base al fatto che il percorso passi vicino a Blinky o a Pinky. Questa analisi non è fine a sé stessa, infatti come abbiamo visto nei capitoli precedenti ogni fantasma ha una sua personalità e quindi anche una diversa pericolosità, per cui è ragionevole, nell'impossibilità di evitarli del tutto, preferire passare in prossimità di quello meno pericoloso.
Per implementare questa nuova caratteristica è sufficiente modificare in

```
return 225;
```

la condizione di controllo sull'array `p_iter`.

```
return 300;
```

quella sull'array `b_iter`.
Ricompiliamo ed eseguiamo e l'output risulta:

```
26-> 25 24 20 14 10 3 :13 :12
```

```
25-> 24 20 14 10 3 :12 :13
24-> 25 26 27 22 19 16 17 11 8 7 6 5 4 3 :15 :16
25-> 26 27 28 29 23 17 11 8 7 6 5 4 3 :14 :19
26-> 21 18 15 14 10 3 :17 :22
21-> 18 19 16 13 12 4 3 :14 :23
18-> 19 16 13 7 6 5 4 3 :15 :17
19-> 18 21 20 14 10 3 :12 :11
18-> 19 16 17 11 8 7 6 5 4 3 :15 :8
19-> 16 13 12 4 3 :14 :7
16-> 13 12 4 3 :20 :6
13-> 12 15 14 10 3 :21 :5
12-> 4 3 :18 :4
4-> 3 :15 :3
```

Come si vede dai risultati qui sopra, alla riga dieci, questa volta, il percorso calcolato passa vicino a Pinky anziché a Blinky in quanto i nodi vicini a quello di Pinky risultano euristicamente meno lontanti dal `goal` rispetto quelli di Blinky (vedi figura 20)

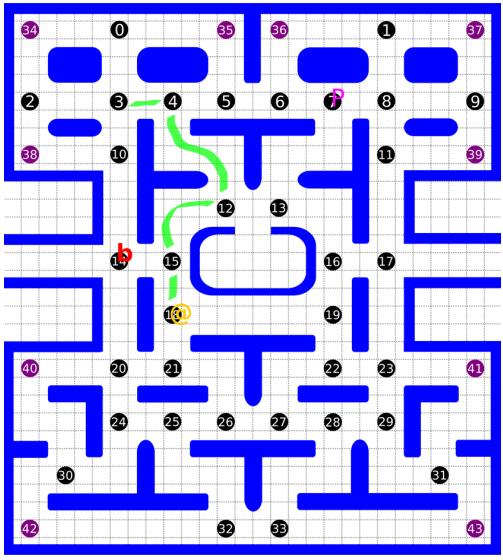


Fig. 20 - Percorso calcolato da A* tra i vertici 18 e 3. Il percorso comprende un vertice contiguo a

quello occupato da Pinky.

Considrazioni

Prima di proseguire con la ricerca della soluzione alla fuga dell'agent nel labirinto, possiamo trarre alcune considerazioni.

La prima è che data la struttura del labirinto non esiste una soluzione che possa basarsi puramente sul grafo. Infatti proviamo ad immaginare che i due fantasmi si fissino sulle posizioni 14 e 7 appena viste. In questa condizione possiamo anche pensare ad implementare la logica dell'agent in modo da modificare il percorso e preferire i vertici *alti*, cioè quelli che vanno dal 17 in su (cioè quelli della parte bassa del labirinto), ma quando la sezione inferiore sarà stata completata, il Pac-Man dovrà dedicarsi alla ricerca delle pillole della sezione superiore e finirà obbligatoriamente vittima dell'attacco di un fantasma, anche se fossero solo due.

La seconda considerazione è che la preparazione di un percorso da seguire diventa una strategia debole nel momento in cui entrano in gioco i fantasmi e costringono l'agent a fuggire. In quella situazione infatti l'agent fuggendo mangia altre pillole che rendono obsoleti certi passaggi del suo percorso predefinito. Continuando a seguire il percorso l'agent perde tempo e si espone inutilmente al rischio di essere attaccato.

Nel prossimo capitolo vediamo come implementare nella piattaforma del gioco le considerazioni svolte.

Una pillola... vale l'altra

Nel capitolo precedente abbiamo intuito un possibile modo per trasferire la logica di fuga dell'agent dentro alla funzione che calcola la stima euristica della distanza.
Il vantaggio di questo approccio è che Pac-Man non si muove alla cieca verso il suo obiettivo per poi invertire la marcia di fronte alla minaccia dei fantasmi, ma piuttosto pianifica dal principio un percorso che lo tenga lontano dai pericoli.

Nel codice di test abbiamo visto che questo approccio può funzionare, ma nel verificarlo abbiamo eseguito alcune semplificazioni che lo hanno astratto dalla piattaforma del gioco. Infatti abbiamo ipotizzato che Pac-Man e i fantasmi si muovessero solo da vertice a vertice senza passare per i corridoi, inoltre abbiamo creato dei percorsi fissi per i fantasmi invece che lasciarli muovere secondo gli algoritmi che ne definiscono le personalità.
Vediamo ora come implementare detta logica nel gioco reale.

Grafo completo del labirinto

Per applicare correttamente la strategia discussa è necessario tener conto che sia Pac-Man che i fantasmi possono trovarsi in una qualsiasi delle celle accessibili e non solo sulle celle che rappresentano un vertice del grafo.
Conviene rivedere la logica con cui sono state divise le celle come vertici e archi e ripensare al labirinto come composto solo da celle-vertice connesse tra loro da *archi virtuali*, cioè archi che vengono percorsi istantaneamente e lungo i quali non è possibile stazionare. In pratica ogni cella *accessibile* diventa un vertice che è collegato a tutte le celle *vicine* che come lei risultano accessibili.

Per rappresentare il grafo possiamo continuare ad utilizzare la struttura `agri_Vertex` e definire riga e colonna di ogni cella accessibile a Pac-Man. Diversamente da prima non lo faremo direttamente analizzando il grafo, ma modificheremo la funzione `collega_tuki_nodi` perché assegni riga e colonna ad ogni vertice in automatico:

```
int k = 0;
  for(int i = 3; i<ALTEZZA-3; i++)
    {
      for(int j = 1; j<LARGHEZZA-1; j++)
        {
          if( oggetto_accessibile(labx[i][j]))
            {
              grafo[k].linea = i;
              grafo[k].columna = j;
              grafo[k].index = k;
              k++;
            }
        }
    }
```

Con il codice qui sopra abbiamo definito tutti i vertici del grafo, ma questi risultano ancora scollegati gli uni dagli altri. Per collegarli tra di loro è necessario iterare su ognuno di essi e verificare a quali delle celle vicine risulta contiguo.

```
for(int k = 0; k<NODI_LAB_POT; k++)
    {
      int r = grafo[k].linea;
      int c = grafo[k].columna;
      if(c>0)
        grafo[k].ianua[SINISTRA] =  trova_vertice(r, c-1);
      if(c<LARGHEZZA - 1)
        grafo[k].ianua[DESTRA] = trova_vertice(r, c+1);
      if(r>0)
        grafo[k].ianua[SU] = trova_vertice(r-1, c);
      if(r<ALTEZZA-1)
        grafo[k].ianua[GIU] = trova_vertice(r+1, c);
    }
```

dove la funzione `trova_vertice` è analoga alla funzione `indice_tuki_nodo_cella` usata nei codici precedenti.

Implementazione dell'agent

Strano ma vero il codice di questo agent è sorprendentemente semplice rispetto quello degli agent visti fin'ora. Infatti la sua logica non prevede alcun meccanismo di fuga, ma solo la scelta *casuale* di un vertice da raggiungere, cioè quello che chiamiamo *goal* e la chiamata continua all'algoritmo A* per conoscere il vertice su cui muoversi.

Sebbene il vertice goal sia scelto casualmente, prima di calcolare il percorso l'agent verifica che in corrispondenza al vertice scelto ci sia una pillola e, in caso contrario, sceglie un altro vertice.
Questo modalità di scelta verrà ridiscussa e migliorata nel prossimo capitolo quando vedremo l'uso dell'alagoritmo *breadth-first search* per migliorare la navigazione dell'agent all'interno del labirinto.

Come nell'esempio svolto con il codice di test, infatti, non è sufficiente eseguire una singola chiamata ad A* per poi *consumare* uno alla volta i vertici del cammino prodotto, perché ad ogni ciclo di gioco i fantasmi possono spostarsi andando in questo modo a modificare quello che sarebbe il cammino ottimale per raggiungere il *goal*. Per questo motivo, pur mantenendo fisso il goal, il cammino viene ricalcolato ad ogni turno chiamando la funzione `agri_astar` e passando come argomento il vertice occupato da Pac-Man e il vertice goal.

Risultati

I risultati di questa implementazione sono piuttosto soddisfacenti, specie se tenuto conto che non è stato implementato nessun algoritmo esplicito per la fuga dai fantasmi.

Per illustrare e discutere le performance di questo agent

abbiamo eseguito duecento partite sfidando rispettivamente 1, 2, 3 e 4 fantasmi per un totale di ottocento partite. Per ogni partita è stato registrato il punteggio totale raggiunto, ricordando che il punteggio di 244 corrisponde al completamento del labirinto. I risultati ottenuti per l'agent `gioca_tuki_pesato` sono stati confrontati con l'agent `gioca_tuki_percorso` ottenuti con le stesse modalità.

Blinky

Lanciando il gioco (`gioca_tiki_pesato`) con il solo fantasma Blinky, l'agent conduce Pac-Man a concludere quasi sistematicamente con successo la partita. Rispetto agli esempi che implementano l'algoritmo di fuga esplicito, vediamo le seguenti differenze:

- Pac-Man non si dirige più spontaneamente verso Blinky per poi cambiare direzione in sua prossimità.
- I percorsi di fuga di Pac-Man non sembrano più *distrarlo* dai suoi vertici obiettivo.
- Pac-Man e Blinky sono mediamente più lontani durante il gioco.
- Pac-Man completa il labirinto in circa metà turni di gioco rispetto a `gioca_tuki_percorso`.

In figura 21 sono riportati gli istogrammi dei risultati delle partite eseguite usando i due diversi agents. Si nota che mentre l'agent *percorso* ha completato con successo tutte le partite, l'agent *pesato* ha una piccola frazione (circa l'8%) di partite non completate con successo.

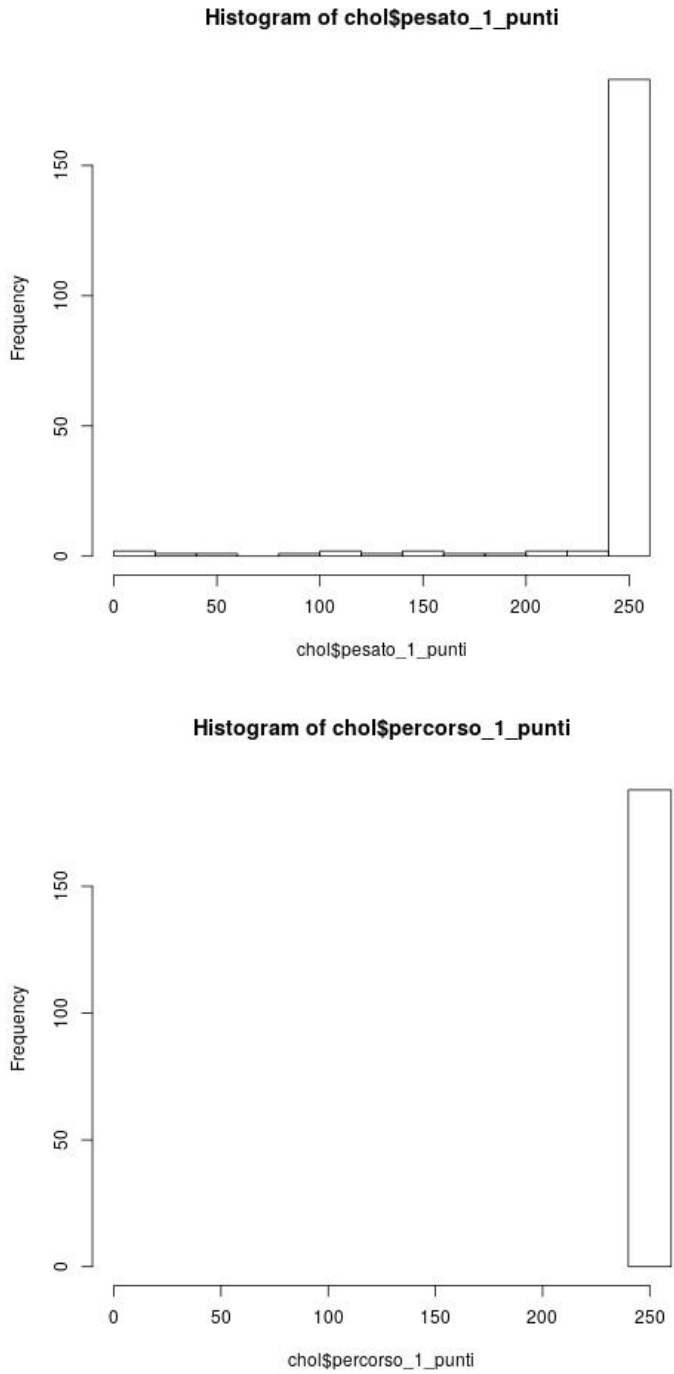
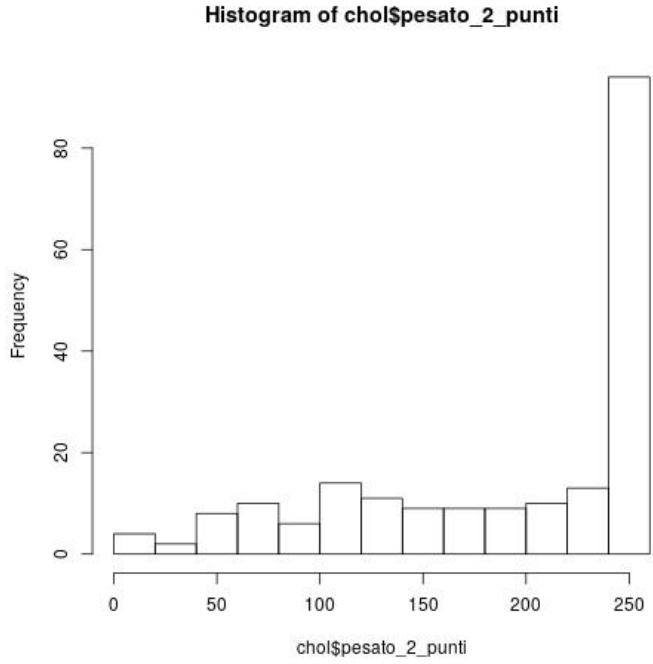


Fig. 21 - Istogramma del punteggio raggiunto giocando contro un singolo fantasma. La figura

sopra si rifersce all'agent *pesato*, quella sotto al *percorso*.

Blinky e Pinky

In presenza della coppia Blinky e Pinky le difficoltà di Pac-Man aumentano, ma qui è dove il nuovo agent si distingue decisamente rispetto al primo.
Le probabilità di successo dell'agent *pesato* sono vicine al 50% e comunque il completamento del labirinto risulta essere l'evento relativamente più probabile rispetto a tutti gli altri punteggi (vedi fig. 22).
Le cose stanno diversamente per l'agent *percorso* che come si vede da figura 22, da come risultato più probabile la cattura di Pac-Man prima che abbia mangiato venti pillole.



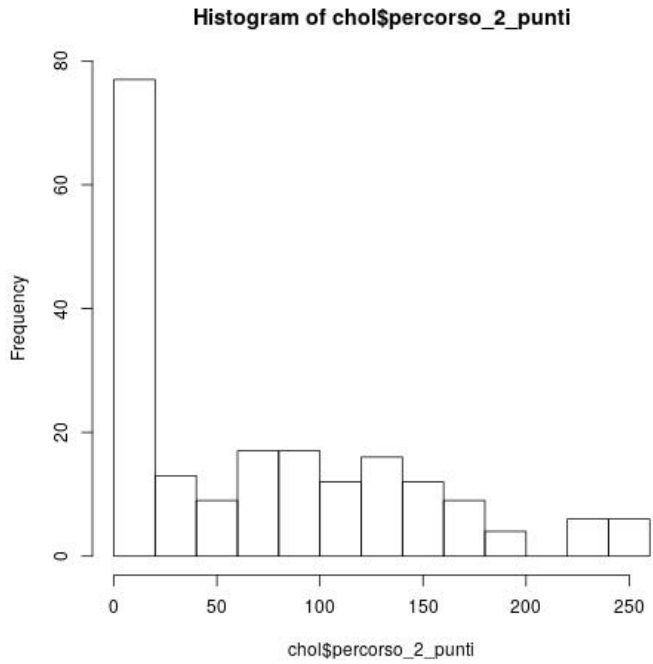


Fig. 22 - Istogramma del punteggio raggiunto giocando contro i fantasmi Blinky e Pinky. La figura sopra si rifersce all'agent *pesato*, quella sotto al *percorso*.

Blinky, Pinky e Inky

L'arrivo di Inky non sconvolge le prestazioni dell'agent *percorso*. Calano le sue possibilità di completare il labirinto (in realtà in nessuna delle duecento partite giocate a 3 fantasmi l'agent lo ha completato), ma la distribuzione dei punteggi rimane pressapoco la stessa ottenuta nelle partite con due soli fantasmi (vedi figura 23).

L'agent *pesato* diversamente da *percorso* è capace di completare il labirinto anche in presenza di tre fantasmi. Come si vede in figura 23 l'agent ha pressapoco la stessa possibiità di vincere che quella di terminare il gioco con un punteggio all'interno di uno qualsiasi degli intervalli di

campionamento. Nello specifico, l'agent ha completato il labirinto 19 volte su 200, cioè circa il 10% delle partite.

Con la presenza di tre fantasmi è interessante notare come avviene la cattura di Pac-Man. Le partite condotte dall'agent *percorso* finiscono spesso con Pac-Man intrappolato tra Blinky e Inky. Questo si spiega se si pensa agli agent che guidano i due fantasmi. Infatti, Inky è scritto per inserirsi proprio tra Pac-Man e Blinky, quindi è naturale che l'azione dei due insieme porti Pac-Man ad esaurire le vie di fuga.
Le cose però vanno diversamene quando è l'agent *pesato* a condurre il gioco che spesso finisce con la cattura di Pac-Man da parte di Pinky.
Questo agent non si comporta come l'altro che reagisce alla prossimità dei fantasmi, ma al contrario pianifica il proprio percorso cercando di evitarli, o meglio pianifica il percorso più leggero, cioè quello che risulta più *breve* tra il vertice su cui si trova Pac-Man e il vertice obiettivo. Per comprendere il perché della fine Pac-Man tra le grinfie di Pinky, anzichè Blinky o Inky come capita con l'altro agent, dobbiamo analizzare proprio l'algoritmo A* alla base dell'agent in combinazione con lo specifico algoritmo che guida Pinky durante il gioco.

Pinky è programmato per puntare avanti quattro celle rispetto a Pac-Man e questo lo porta spesso ad essere su un percorso diverso rispetto a Blinky e Inky che invece tendono a viaggiare appaiati.
Si capisce che Pac-Man tende quindi a trovarsi tra due poli, uno in cui è presente Pinky e l'altro in cui sono presenti e vicini Blinky e Inky. Il loro peso viene quindi sommato e il percorso che punta verso Pinky preferito a quello che punta agli altri due fantasmi.

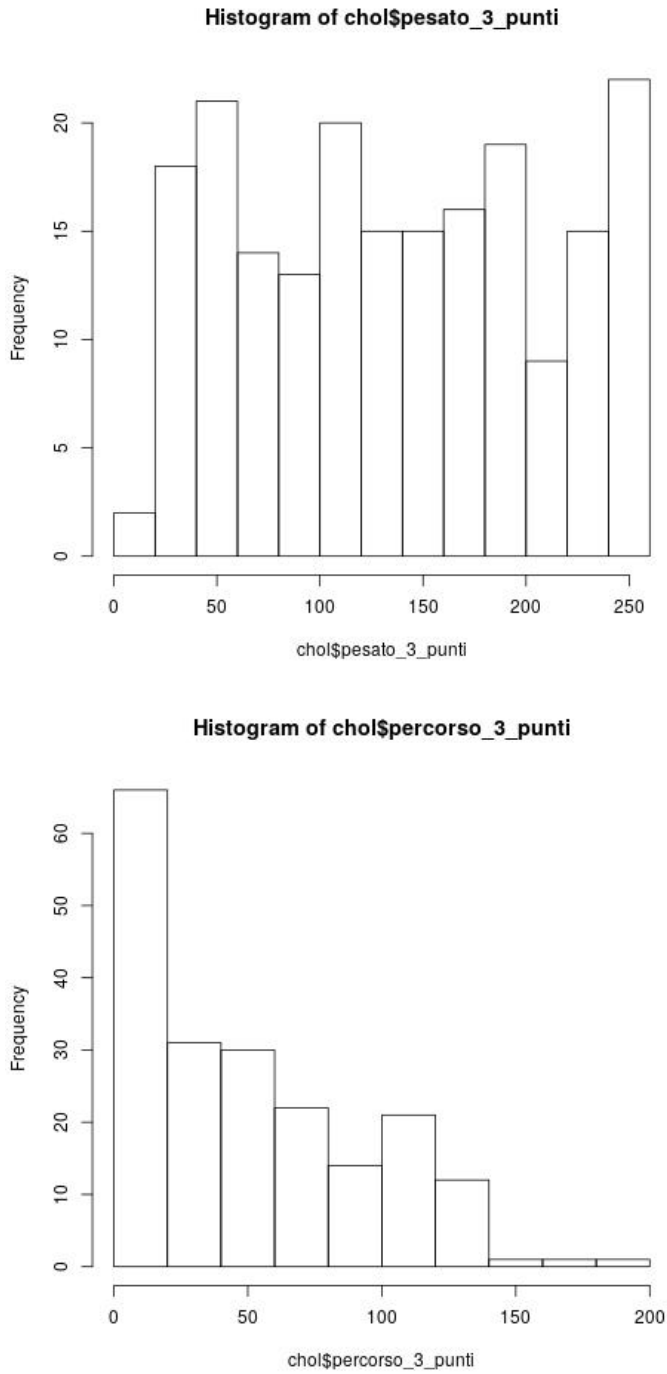
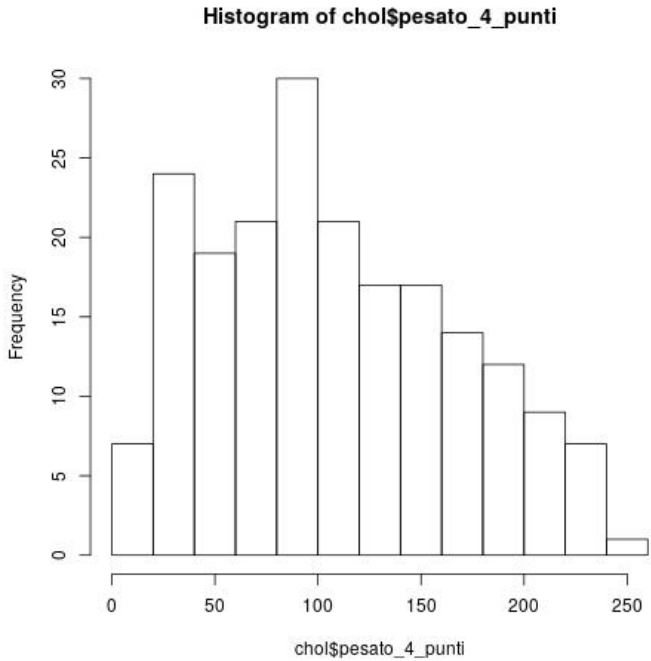


Fig. 23 - Istogramma del punteggio raggiunto giocando contro i fantasmi Blinky e Pinky. La

figura sopra si rifersce all'agent *pesato*, quella sotto al *percorso*.

Blinky, Pinky, Inky e Clyde

Il compito di questo agent di fronte all'intero ghost team diventa superiore alle sue possibilità, infatti di 200 partite ne vince solo una, però i suoi risultati sono nettamente migliori rispetto all'agent *percorso* che praticamente finisce più della metà delle partite con un punteggio inferiore a 30 (vedi figura 24).



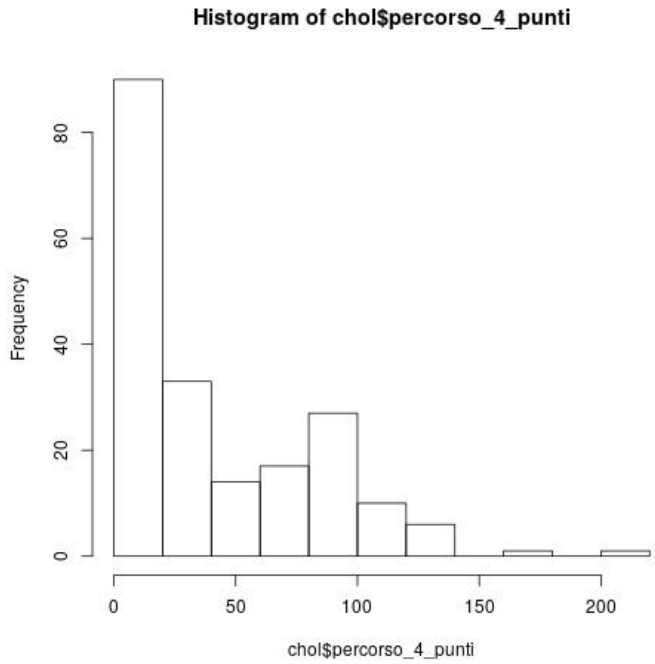


Fig. 24 - Istogramma del punteggio raggiunto giocando contro i fantasmi Blinky e Pinky. La figura sopra si rifersce all'agent *pesato*, quella sotto al *percorso*.

Considerazioni

La nuova implementazione della funzione per la stima euristica della distanza ha contribuito in modo evidente alle capacità di fuga di Pac-Man. Il risultato è senz'altro interessante in quanto la fuga *emerge* dall'algoritmo senza essere stata specificatamente implementata.
Questo agent ha però ancora due limiti importanti. Il primo è che la scelta del vertice *goal* è basata sulla funzione rand che, sebbene venga corretta in caso di cella vuota, non rappresenta comunque una scelta ragionata dell'obiettivo. Il secondo è che anche questo agent non sfrutta la conoscenza delle *abitudini* dei fantasmi.

Questo punto merita un approfondimento. Anzi tutto vediamo che in linea di principio non è possibile definire un agent che vinca a priori se non si stabiliscono alcune regole del gioco che infatti Toru Iwatani ha saputo inserire sapientemente:

- Le pillole energetiche permettono a Pac-Man di mangiare i fantasmi
- I fantasmi non possono stazionare in una cella in attesa di Pac-Man

La prima regola aiuta molto il completamento del gioco, ma la seconda è pressoché indispensabile. I vertici del grafo del labirinto di Pac-Man hanno grado massimo 4. Considerando che i fantasmi sono 4 è facile capire come possano organizzarsi per presidiare un vertice e quindi impedire il completamento del gioco o comunque bloccare Pac-Man in un corridoio o in un vertice. Questo ci fa capire che la fuga non può essere garantita senza il rispetto di queste regole.

Anche se i fantasmi si organizzassero in un ronda attorno ad una delle isole il gioco sarebbe irrisolvibile. Da questo segue che la vittoria sul ghost team non può essere assicurata nel labirinto del Pac-Man se non si modella l'agent di Pac-Man sull'implementazione specifica degli agent dei fantasmi.
Questo obiettivo può essere raggiunto sfruttando la conoscenza di detti agent che si ha a priori, in quanto si conoscono le loro regole interne, oppure programmando un agent che apprenda partita dopo partita.
Entrambe queste soluzioni sono però indipendenti dalla teoria dei grafi e vanno al di fuori dell'obiettivo di questo libro.

Box domande n.6

1. Cos'è il grado minimo di un grafo?
    a. Il numero minimo di vertici su è necessario passare per coprire tutti gli archi
    b. I numero minimo di vertici che formano un anello all'interno del grafo
    c. Il numero minimo di archi entranti (o uscenti) di ogni vertice
4. Perché nell'agent `percorso` non è sufficiente calcolare il percorso una volta per tutte a *desing-time*, ma è necessario ricalcolarlo od ogni turno durante l'esecuzione?
    a. Perché durante il gioco i fantasmini si muovono andando a cambiare i pesi dei possibili percorsi
    b. Perché durante il gioco Pac-Man si muove cambiando le sue coordinate
    c. Perché la funzione euristica necessita un aggiornamento continuo
4. Qual è la differenza tra un meccanismo di fuga basato sul controllo diretto e uno basato sull'euristica pesata?
    a. Il controllo diretto, quando Pac-Man si trova ad una certa distanza dal fantasma, lo fa scappare allontanandolo dal percorso che stava seguendo, mentre il controllo basato sull'euristica pesata consiste in una continua riconfigurazione del percorso che consenta di evitare i fantasmi

b. Il controllo diretto è l'equivalente di giocare con un JoyStick. L'agent non pianifica un percorso ma stabilisce ad ogni turno di gioco la cella su cui far spostare il Pac-Man. L'agent che implementa l'euristica pesata, invece, stabilisce un percorso e lo riconfigura durante il gioco.
c. Il controllo diretto fa parte dell'euristica pesata. E' l'operazione con cui l'agent, prima di ricalcolare il peso del percorso, identifica le posizioni dei fantasmi

4. Perché il Pac-Man, guidato dall'agent `percorso` ed inseguito da tre fantasmi (Blinky, Pynky e Inky), viene più facilmente mangiato da Pinky che dagli altri due?
    a. Perché Pinky punta sempre alla cella che si trova 4 posizioni davanti a Pac-Man, inserendosi all'interno di un percorso per cui lui ha già calcolato il peso
    b. Perché tra tutti i fantasmi Pinky è quello meno "pesante"
    c. Perché Blinky e Inky, si muovono insieme, vicini e, nel caso Pac-Man si trovi a dover scegliere se dirigersi verso loro due o verso Pinky, sceglierà la direzione di Pinky, in quanto meno "pesante"

La pillola più vicina

In questo capitolo ci occupiamo della scelta ragionata del vertice *goal* che è l'ultimo aspetto rimasto ancora *naive* dell'implementazione dell'agent di Pac-Man.

Prima... respira

Per implementare un agent che scelga in modo *intelligente* il vertice verso cui dirigersi abbiamo bisogno di un nuovo algoritmo che fa parte dell'armamentario di chi si occupa di grafi, questo è il *Breadth-first search* (BFS) che insieme all'algoritmo *Depth-first search* (DFS) è un algoritmo di attraversamento e ricerca attraverso un grafo.

L'idea alla base del BFS è di ricercare all'interno di un grafo valutando prima tutti i vertici vicini al vertice di partenza (start), poi, se tra loro nessuno corrisponde al criterio di ricerca, passare al livello successivo, cioè a valutare i vertici vicini dei vicini e così via.
Nel nostro caso, il criterio di ricerca è la presenza di una pillola. In pratica, dato il vertice su cui si trova Pac-Man, vogliamo trovare il primo vertice che contiene una pillola, cioè non vuoto.

In figura 25 è rappresentato Pac-Man in una regione vuota di pillole. Lo scopo è quello di dirigerlo verso la pillola più vicina. L'algoritmo BFS valuta prima i vicini prossimi al vertice di Pac-Man che sono indicati con il numero 1, poi i vertici direttamente collegati ai suoi vicini, indicati con il numero 2. La ricerca dell'algoritmo continua finchè non viene trovato il primo vertice con una pillola.

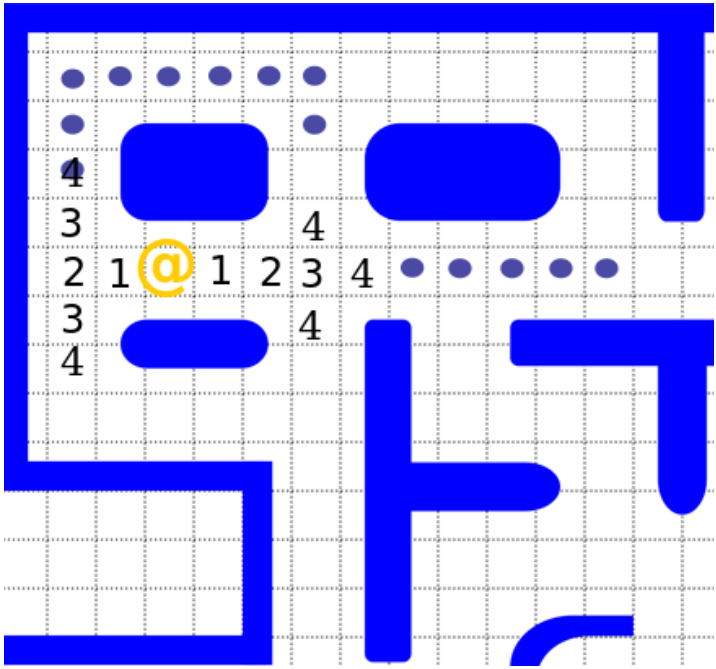


Fig. 25 - I vertici accessibili vicini a quello in cui si trova Pac-Man sono indicati dal numero 1 che indica il livello di prossimità. I vicini dei vertici di livello 1 sono indicati dal numero 2 e via dicendo, i vicini dei vertici di livello 2 sono indicati con un 3 e via dicendo. La prima pillola si trova su un vertice di livello 4.

Il BFS si distinge dal DFS che invece preso uno dei vicini del vertice start, continua la ricerca attraverso i vertici collegati finché non trova un vertice corrispondente al criterio di ricerca, se non lo trova, passa al prossimo vicino di livello 1. Questo secondo approccio non è ottimale in un labirinto completamente connesso come quello di Pac-Man in quanto dato il vertice start, la ricerca attraverso il primo dei vicini di livello 1 risulterebbe sempre comunque esaustiva (figura 26).

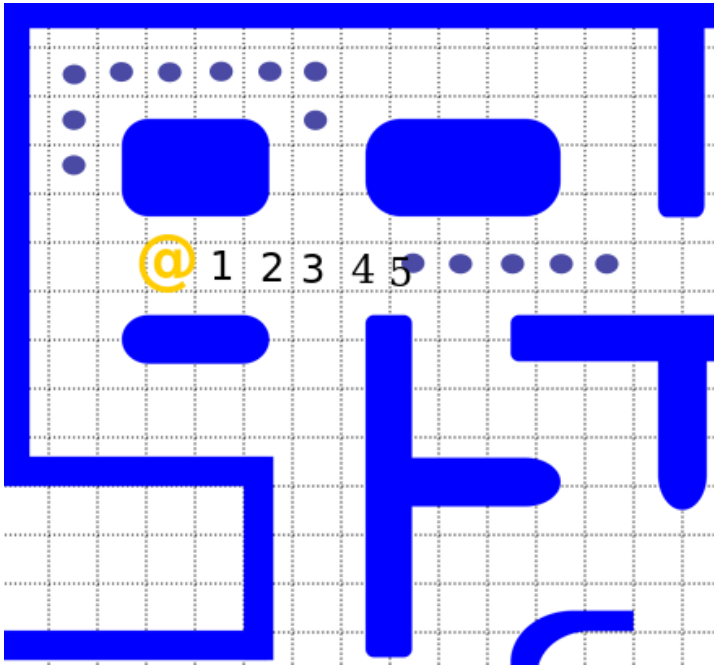


Fig. 26 - I vertici accessibili vicini a quello in cui si trova Pac-Man sono indicati dal numero 1 che indica il livello di prossimità. I vicini dei vertici di livello 1 sono indicati dal numero 2 e via dicendo, i vicini dei vertici di livello 2 sono indicati con un 3 e via dicendo. La prima pillola si trova su un vertice di livello 5.

Implementazione BFS

L'implementazione dell'algoritmo che proponiamo è funzionale allo scopo dell'agent e risulta più semplice di quella normalmente proposta. Infatti useremo il BFS non per calcolare il percorso verso il primo vertice *papabile* (cioè con la pillola) ma solo per conoscerne l'indice, mentre poi useremo l'algoritmo A* precedentemente sviluppato per calcolare il percorso dalla posizione corrente ad esso.

```
int agri_breadthfirstsearch(int start,
                    agri_Vertex * agri_Vertices_Colligati,
                    int (*visitatus)(int ),
```

```
                    int nmembri
                    );
```

Come si vede dal prototipo la funzione non torna un path, ma un intero che rappresenta l'indice del vertice trovato.

L'implementazione in sé stessa è molto simile a quella dell'A*.
Anzitutto il vertice *start* viene inserito nella lista usata per implementare la coda dei vertici che sono stati valutati:

```
Ordo_insero_nodus(&candidati,start,1);
```

Il vertice viene poi estratto dalla coda e vengono ricercati tutti i suoi vicini che a loro volta vengono inseriti nella coda:

```
int corrente = Ordo_pop(&candidati);

for(int i=0; i<PORTE; i++)
       {
         vicino[i] = agri_Vertices_Colligati[corrente].ianu
       }

     for(int i =0; i<PORTE; i++)
       {
         int iv = vicino[i];

         if(iv == -1)continue;

         precedente[iv] = corrente;

         Ordo_insero_nodus(&candidati,iv,1);
       }
```

Per ognuno dei vicini viene quindi ripetuta la stessa procedura:

- Estrazione della coda
- Verifica della presenza della pillola
- Ricerca dei vicini con inserimento nella coda

finché la coda non risulta vuota oppure finché in uno dei vertici viene trovata la pillola.

Implementazione dell'agent *respiro*

Il codice completo dell'agent che stiamo sviluppando è riportato nel listato `gioca_tuki_respiro.c`. La differenza fondamentale rispetto all'agent *pesato* si trova nella scelta del vertice *goal*.
Nell'agent precedente esso veniva scelto in modo pseudo-casuale

```
do{
        vertice_goal = (double)rand()/((double)RAND_MAX)*NOD
  } while(labx[grafo[vertice_goal].linea][grafo[vertice_goal
```

tra tutti i vertici del grafo, salvo poi filtrare il risultato con solo quelli in cui è presente la pillola.
Nell'agent presente invece il punto di forza è proprio nella scelta del *goal* che risulterà sempre una scelta ragionata

```
 vertice_goal = agri_breadthfirstsearch
        (vertice_corrente,
         grafo,
         visitatus,
         NODI_LAB_POT
         );
```

Test dell'agent

Una volta editato e compilato l'agent secondo la solita procedura già vita in precedenza, il primo test da condurre è quello di Pac-Man in assenza di fantasmi.

Lanciando l'eseguibile non portà sfuggire la naturalezza con la quale Pac-Man completerà il labirinto senza scatti e interruzioni lungo il cammino.
Un risultato simile è stato ottenuto anche con l'agent *percorso*, ma in quel caso il percorso era prestabilito a *design-time* quindi era naturale che in assenza di fantasmi risultasse scorrevole.

Le sorprese arrivano quando il gioco viene lanciato con la

presenza di uno o più fantasmi, infatti già in presenza di un solo fantasma vediamo che la probabilità che l'agent completi il labirinto è molto bassa nonostante si usi ancora l'algoritmo A* pesato per la pianificazione del percorso. Perché succede questo?
La risposta è invero piuttosto ovvia. La scelta del vertice tramite BFS comporta come prima conseguenza che una volta che Pac-Man ha imboccato un corridoio in cui sono presenti le pillole, lo completerà tutto passando iterativamente dalla posizione in cui si trova al vertice *goal* corrispondente esattamente alla cella successiva a quella che sta occupando. In pratica il risultato della chiamata ad BFS è il vertice successivo a quello in cui si trova Pac-Man. Dato questo, l'uso dell'A* con la funzione euristica pesata risulta inutile finché nel percorso Pac-Man non incontra un punto di discontinuità (per esempio un corridoio già svuotato) e in questo caso l'A* ha l'opportunità di pianificare un percorso che tenda ad escludere i fantasmi.

Nel prossimo capitolo vediamo come preservare i vantaggi acquisiti con l'agent *pesato* sfruttandoli insieme al'agent *respiro*.

Box domande n.7

1. In che modo A* e BFS collaborano al raggiungimento della cella contenente la pillola più vicina
    a. BFS calcola i possibili percorsi per raggiungere la cella e A* sceglie il più breve
    b. BFS individua la cella, A* calcola il percorso più breve per raggiungerla
    c. A* e BFS calcolano entrambi un percorso per raggiungere la cella e viene scelto il più breve
4. Cosa resitiuisce la funzione `agri_breadthfirstsearch`?
    a. L'indice della prossima cella da raggiungere
    b. La sequenza di celle da percorrere
    c. L'indice della prossima cella per cui passare per raggiungere la cella bersaglio
4. Perché l'uso di A* con l'euristica pesata risulta inutile nel percorso finché Pac-Man non si trova in una cella che non abbia pillole confinanti?
    a. Perché BFS non chiama A* quando la cella obiettivo è una cella limitrofa
    b. Perché A* considera solo il peso del percorso che calcola dalla cella di partenza alla cella obiettivo. Sa la cella obiettivo è, ad ogni turno, la cella immediatamente limitrofa, non si accorgerà se si sta dirigendo o meno in bocca a un fantasma.
    c. Perché in presenza del BFS l'A* prevale

sull'euristica pesata

Il respiro... pesato

In questo capitolo non introduciamo nessun nuovo concetto, ma vediamo come usare l'algoritmo BFS in un modo diverso dal suo uso comune. Nel capitolo precedente lo abbiamo usato per scoprire il vertice (cioè la cella) più vicina a quella di Pac-Man con presente una pillola.
Come abbiamo visto in precedenza il labirinto di Pac-Man non permette un cammino euleriano e quindi in tutti i casi, anche in assenza di fantasmi, non è possibile completare il gioco senza mai *staccare*, cioè senza dover ritornare lungo un percorso già esplorato. Ribadiamo questo concetto perché spiega ulteriormente il ruolo del BFS nel realizzare una partita dove Pac-Man *scorre* in modo abbastanza naturale, senza eccedere nel *vagabondaggio* alla ricerca di tutte li pillole.

La richiesta di *scorrevolezza* però abbiamo visto essere in conflitto con l'esigenza di fuggire, in quato questa comporta per forza il dover abbandonare il cammino prescelto per evitare gli attacchi dei fantasmi. Inoltre, se la fuga è basata sul calcolo del cammino pesato usando l'A*, questi non ha modo di espletare la sua funzione in quanto nella maggior parte dei casi la cella *goal* è sempre quella innanzi a Pac-Man. Quindi l'algoritmo BFS e l'A* insieme, non fuzionano per l'agent di Pac-Man.

Ricercare i fantasmi

Per dare qualche chance in più a Pac-Man di completare la partita è necessario che l'agent eviti gli attacchi del ghost team quando i fantasmi sono nei paraggi. All'inizio del libro, la prima soluzone che abbiamo intrapreso in questa direzione

è stato il controllo delle celle vicine a quella di Pac-Man con il relativo cambio di direzione in concomitanza alla presenza di un fantasma. Come si è visto però questo approccio alla fuga funziona egregiamente quando c'è solo un fantasma, ma risulta poco efficace già quando i fantasmi sono due. Inoltre ha poco a che fare con la teoria dei grafi, per cui in ogni caso è una strategia che una volta vista non implementiamo nuovamente.

In questo paragrafo vogliamo invece implementare una nuova strategia *evitante* che porti l'agent ad avere consapevolezza della presenza dei fantasmi in modo da poter attuare una strategia difensiva.

L'idea è quella di usare l'algoritmo BFS in modo alternativo, per eseguire una ricerca *breath first* dei fantasmi attorno a Pac-Man.
A questo scopo copiamo il nostro stesso codice dalla libreria `agri` e creiamo una funzione utente basata sul principio del BFS ma con un compito più specifico: ricercare i fantasmi in prossimità di Pac-Man proseguendo la ricerca fino ad una certa prodondità (o livello). Se il fantasma viene trovato la funzione ritorna il livello a cui è stato trovato il fantasma, in modo simile ad un radar.
Basandoci su detto livello l'agent decide se abbandonare la ricerca delle pillole per darsi alla fuga. In pratica il livello corrisponde alla distanza del fantasma più vicino a Pac-Man il quale può stabilire la propria asticella di allarme e decidere di conseguenza (vedi figura 27).

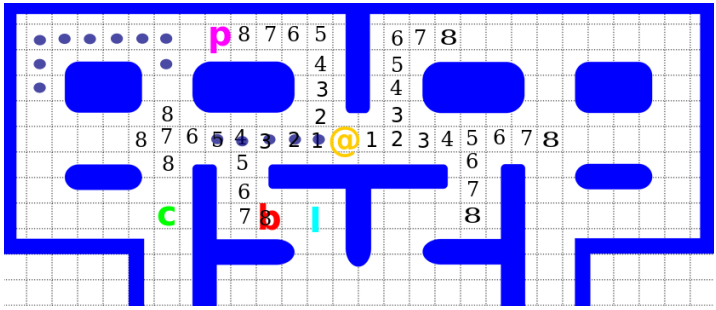


Fig. 27 - Pac-Man e il ghost team sono nel labirinto. I numeri dentro alle celle corrispondono alla distanza misurata in celle dalla posizione di Pac-Man alla cella corrispondente.

Chiamiamo detta funzione `phantasmatis_presentia` il cui prototipo è analogo a quello della `agri_breadthfirstsearch`:

```
int phantasmatis_presentia(
        int start,
        agri_Vertex * agri_Vertices_Colligati,
        int (*fantasma_presente)(int ),
        int nmembri
        )
```

La `phantasmatis_presentia` viene chiamata dall'agent fintanto che non viene individuato un vertice cella raggiungibile attraverso un percorso sicuro:

```
int tent = 0;
while(sicuro<DISTANZA_SICUREZZA)
{
  do{
    tent++;
    vertice_goal =
    (double)rand()/((double)RAND_MAX)*NODI_LAB_POT;
          }
    while(
      labx[grafo[vertice_goal].linea]
      [grafo[vertice_goal].columna] == 'J' &&
      tent<TENTATIVI
      );
    vertice_goal =
      evita_casa_fantasmi(vertice_goal);
    sicuro =
      phantasmatis_presentia
      (vertice_goal, grafo,
```

```
        phantasmatis,
        NODI_LAB_POT);
}
```

Test e verifica

L'agent completo che implementa la logica di fuga dai fantasmi è presentato nel listato `gioca_tuki_evita.c`. Dopo averlo editato e compilato secondo le modalità già presentate si può procedere alla verifica delle sue performance partendo dal gioco senza ghost team per poi aumentare gradatamente il numero di fantasmi.

Come è naturale aspettarsi, in assenza di fantasmi l'agent *evita* mostra lo stesso comportamento dell'agent *respiro*. Inserendo nel gioco il primo fantasma, l'agent riesce sempre a completare il labirinto continuando a mostrare un andamento *scorrevole* e naturale nella scelta del percorso.
Con due fantasmi le difficoltà dell'agent aumentano ma esso continua ad esibire fluidità di movimento e un alta percentuale di successo nel completamento delle partite.
Con tre e quattro fantasmi l'agent continua a vincere alcune partite, ma la percentuale scende significativamente. La cosa veramente notevole è che si manifesta spontaneamente un comportamento *intelligente* dell'agent che sembra prevedere le mosse dei fantasmi e sganciarsi dal percorso che stava percorrendo per sfuggire loro come se fosse guidata da un essere umano o da un sistema di AI.

Ulteriori applicazioni

La funzione `phantasmatis_presentia` può trovare anche un'altra applicazione nel gioco. Essa può essere usata per dirigere Pac-Man verso i fantasmini e quindi cacciarli quando sono nello stato *frightened* (impaurito), cioè sono blu.

Ovviamente si dovrà anche completare la gestione dello stato *frightened* che in questa piattaforma non è gestito in modo completo per motivi di semplicità espositiva e di spazio.

CONCLUSIONE

In questo libro abbiamo cercato di aiutare Pac-Man a ottimizzare il suo tempo cercando di attraversare il labirinto con "consapevolezza" e anche sfuggendo ai fantasmi che lo inseguono da quarant'anni.
Abbiamo visto che l'uso di un grafo ci consente di ragionare su basi algoritmiche su questi problemi e di trovare soluzioni più efficienti. Inoltre, l'uso di grafi e algoritmi sui grafi consente di ragionare e sviluppare soluzioni indipendenti o quasi indipendenti dalla forma specifica del labirinto.

Questa lettura ha un doppio valore. Da un lato, abbiamo incontrato alcuni concetti teorici dell'informatica e abbiamo visto come lavorare con loro e come sviluppare nuovi algoritmi. Dall'altro, come è stato detto, i problemi di Pac-Man sono in realtà anche i problemi di diversi agenti (spesso robot e droni) che devono muoversi nel mondo reale. I grafi e gli algoritmi sui grafi sono oggetto di studio crescente per la ricerca di soluzioni performanti nel campo della logistica e in altri settori produttivi e scientifici.

APPENDICE: CODICE COMPLETO DELL'MVC E DEI FANTASMI

Listato tuki_controllo.c

```
/**
File: tuki_controllo.c
 */

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <locale.h>

#include "tuki5_modello.h"
#include "tuki5_visore.h"

/* MACRO  */
#define MAJOR    0
#define MINOR    1
#define RELEASE  1

/* PROTOTIPI  */
direzione gioca_tuki(posizioni p, oggetto** lab);
direzione gioca_blinky(posizioni p, oggetto** lab);
direzione gioca_inky(posizioni p, oggetto** lab);
direzione gioca_pinky(posizioni p, oggetto** lab);
direzione gioca_clyde(posizioni p, oggetto** lab);

int main(int argc, char **argv)
{
  /*
    ITA: numero di fantasmi nel labirinto (default 4)
    ENG: number of ghosts in the maze
   */
  int n_fantasmi = 4;
  if(argc>1)
    {
      n_fantasmi = atoi(argv[1]);
      if(n_fantasmi>4) n_fantasmi = 4;
    }

  /* Presentazione */
  view_init();
  //view_presentazione();

  /* Inizio gioco */
  int inizio=  mdl_genera_campo();

  if(!inizio) exit(1);

  /* Labirinto */
  oggetto ** lab = mdl_campo();
  view_labirinto(lab);

  /**
   * START GAME
   */
  int r=1;
  while(r){

    posizioni p=mdl_posizioni();

    if(lab==0) exit(1);

    direzione t = gioca_tuki(p, lab);

    direzione b = gioca_blinky(p, lab);

    direzione i = gioca_inky(p, lab);

    direzione pi = gioca_pinky(p, lab);
```

```
    direzione c = gioca_clyde(p, lab);

    char go;

    if(n_fantasmi == 0)
      go = mdl_passo(t,FERMO,FERMO,FERMO,FERMO);
    if(n_fantasmi == 1)
      go = mdl_passo(t,b,FERMO,FERMO,FERMO);
    if(n_fantasmi == 2)
      go = mdl_passo(t,b,i,FERMO,FERMO);
    if(n_fantasmi == 3)
      go = mdl_passo(t,b,i,pi,FERMO);
    if(n_fantasmi == 4)
      go = mdl_passo(t,b,i,pi,c);

    int pnt = mdl_punteggio();
    char inblu=mdl_superpacman();

    /** mostra campo e giocatori */
    r+=1;
    p=mdl_posizioni();
    view_punteggio(pnt);
    view_giocatori(p,lab,inblu);
    delay(DELAY);

    /** Pacman è stato mangiato **/
    if(go==0)
      {
        view_mangiato(p);
        view_gameover("GAME OVER");
        exit(0);
      }

    /** Pacman ha finito le pastiglie 245 */
    if(pnt == 244)
      {
        view_gameover("Hai vinto!");
        exit(0);
      }
   }
  /*end of the game*/
}
```

Listato tuki_modello.c

```
/*_____________________________________
* FILE: tuki5_modello.c
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "tuki5_modello.h"

/* stato di Tuki */
char sppc = 0;
int conto_rovescia = 10;

/* punti */
int punti = 0;

/* il campo di battaglia */
oggetto  campo[ALTEZZA][LARGHEZZA]={
  {J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J},
  {J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J},
  {A,E,E,E,E,E,E,E,E,E,E,E,E,I,L,E,E,E,E,E,E,E,E,E,B},
  {F,U,U,U,U,U,U,U,U,U,U,U,U,S,S,U,U,U,U,U,U,U,U,U,U,X},
  {F,U,G,T,T,H,U,G,T,T,T,H,U,S,S,U,G,T,T,T,H,U,G,T,T,H,U,X},
  {F,V,S,J,J,S,U,S,J,J,J,S,U,S,S,U,S,J,J,J,S,U,S,J,J,S,V,X},
  {F,U,W,T,T,Y,U,W,T,T,T,Y,U,W,Y,U,W,T,T,T,Y,U,W,T,T,Y,U,X},
  {F,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,X},
  {F,U,G,T,T,H,U,G,H,U,G,T,T,T,T,T,H,U,G,H,U,G,T,T,H,U,X},
  {F,U,W,T,T,Y,U,S,S,U,W,T,T,H,G,T,T,Y,U,S,S,U,W,T,T,Y,U,X},
  {F,U,U,U,U,U,U,S,S,U,U,U,U,S,S,U,U,U,S,S,U,U,U,U,U,X},
  {C,Z,Z,Z,Z,B,U,S,W,T,T,H,J,S,S,J,G,T,T,Y,S,U,A,Z,Z,Z,Z,D},
  {J,J,J,J,J,F,U,S,G,T,T,Y,J,W,Y,J,W,T,T,H,S,U,X,J,J,J,J,J},
  {J,J,J,J,J,F,U,S,S,J,J,J,J,J,J,J,J,J,S,S,U,X,J,J,J,J,J},
  {J,J,J,J,J,F,U,S,S,J,A,E,E,J,J,E,E,B,J,S,S,U,X,J,J,J,J,J},
```

```
  {Z,Z,Z,Z,Z,D,U,W,Y,J,F,J,J,J,J,J,J,X,J,W,Y,U,C,E,E,E,E,E},
  {J,J,J,J,J,J,U,J,J,J,F,J,J,J,J,J,J,X,J,J,J,U,J,J,J,J,J},
  {E,E,E,E,E,B,U,G,H,J,F,J,J,J,J,J,J,X,J,G,H,U,A,Z,Z,Z,Z,Z},
  {J,J,J,J,J,F,U,S,S,J,C,E,E,E,E,E,E,D,J,S,S,U,X,J,J,J,J,J},
  {J,J,J,J,J,F,U,S,S,J,J,J,J,J,J,J,J,J,J,S,S,U,X,J,J,J,J,J},
  {J,J,J,J,J,F,U,S,S,J,G,T,T,T,T,T,H,J,S,S,U,X,J,J,J,J,J},
  {A,E,E,E,E,D,U,W,Y,J,W,T,T,H,G,T,T,Y,J,W,Y,U,C,E,E,E,E,B},
  {F,U,U,U,U,U,U,U,U,U,U,U,S,S,U,U,U,U,U,U,U,U,U,U,U,U,X},
  {F,U,G,T,T,H,U,G,T,T,T,H,U,S,S,U,G,T,T,T,H,U,G,T,T,H,U,X},
  {F,U,W,T,H,S,U,W,T,T,T,Y,U,W,Y,U,W,T,T,T,Y,U,S,G,T,Y,U,X},
  {F,V,U,U,S,S,U,U,U,U,U,U,U,J,J,U,U,U,U,U,U,U,S,S,U,U,V,X},
  {O,T,H,U,S,S,U,G,H,U,G,T,T,T,T,T,H,U,G,H,U,S,S,U,G,T,Q},
  {P,T,Y,U,W,Y,U,S,S,U,W,T,T,H,G,T,T,Y,U,S,S,U,W,Y,U,W,T,R},
  {F,U,U,U,U,U,U,S,S,U,U,U,U,S,S,U,U,U,U,S,S,U,U,U,U,U,X},
  {F,U,G,T,T,T,T,Y,W,T,T,H,U,S,S,U,G,T,T,Y,W,T,T,T,H,U,X},
  {F,U,W,T,T,T,T,T,T,T,Y,U,W,Y,U,W,T,T,T,T,T,T,T,Y,U,X},
  {F,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,X},
  {C,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,D},
  {J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J},
  {J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J}
};
oggetto **copia_campo;

/* Numero di trifogli attivi */
int n_trifogli;

/* Posizione giocatori */
posizioni g;

int mangia()
{
  punti+=1;
  return punti;
}

void superpacman(char si_no)
{
  if(si_no)
    {
      conto_rovescia=30;
      sppc=1;
    }
  else
    sppc=0;
}

/*_____________________________________________________
 *|
 *| Se inverso=1, ogni direzione è inverita
 *| srrve ad eseguire il rollback di un movimento
 */
void muovi(direzione d, int *x,int *y,char inverso)
{
  if(!inverso)
    {
      if(d==SU) (*y)=*y-1;
      if(d==GIU) (*y)=*y+1;
      if(d==SINISTRA) (*x)=*x-1;
      if(d==DESTRA) (*x)=*x+1;
    }
  else
    {
      if(d==SU) (*y)=*y+1;
      if(d==GIU) (*y)=*y-1;
      if(d==SINISTRA) (*x)=*x+1;
      if(d==DESTRA) (*x)=*x-1;
    }
}

char mdl_superpacman()
{
  return sppc;
}

/*_____________________________________________________
 *|
 *| Verifica se riga e colonna di pacman e di uno dei
 *| fantasmi coincido. Non abilitato se pacman è in
 *| modalità superpacman (pillola)
 */
int rileva_collisione()
{
  if(sppc == 1) return 0;
```

```
  //-verifica collisione tuki-fantasma
  if(g.tuki_x == g.blinky_x && g.tuki_y == g.blinky_y)
    {
      return 1;
    }
  if(g.tuki_x == g.inky_x && g.tuki_y == g.inky_y)
    {
      return 1;
    }
  if(g.tuki_x == g.pinky_x && g.tuki_y == g.pinky_y)
    {
      return 1;
    }
  if(g.tuki_x == g.clyde_x && g.tuki_y == g.clyde_y)
    {
      return 1;
    }

  return 0;
}

int mdl_passo
(direzione tuki,
 direzione blinky,
 direzione inky,
 direzione pinky,
 direzione clyde)
{
  //-verifica se la pillola è attiva
  char sp=mdl_superpacman();
  if(sp)
    {
      if(conto_rovescia==0)
        superpacman(0);
      else
        conto_rovescia-=1;
    }

  int collisione=0;

  //-Modifica le coordinate
  muovi(tuki,&g.tuki_x,&g.tuki_y,0);

  collisione = rileva_collisione();

  muovi(blinky,&g.blinky_x,&g.blinky_y,0);

  collisione = collisione || rileva_collisione();

  muovi(inky,&g.inky_x,&g.inky_y,0);

  collisione = collisione || rileva_collisione();

  muovi(pinky,&g.pinky_x,&g.pinky_y,0);

  collisione = collisione || rileva_collisione();

  muovi(clyde,&g.clyde_x,&g.clyde_y,0);

  collisione = collisione || rileva_collisione();

  //-Verifica collisione muro
  if(campo[g.tuki_y][g.tuki_x] != J &&
     campo[g.tuki_y][g.tuki_x] != U &&
     campo[g.tuki_y][g.tuki_x] != V)
    {
      muovi(tuki,&g.tuki_x,&g.tuki_y,1);
      collisione = collisione || rileva_collisione();
    }

  if(campo[g.blinky_y][g.blinky_x] != J &&
     campo[g.blinky_y][g.blinky_x] != U &&
     campo[g.blinky_y][g.blinky_x] != V)
    {
      muovi(blinky,&g.blinky_x,&g.blinky_y,1);
    }

   if(campo[g.inky_y][g.inky_x] != J &&
      campo[g.inky_y][g.inky_x] != U &&
      campo[g.inky_y][g.inky_x] != V)
    {
      muovi(inky,&g.inky_x,&g.inky_y,1);
```

```
    }

    if(campo[g.pinky_y][g.pinky_x] != J &&
       campo[g.pinky_y][g.pinky_x] != U &&
       campo[g.pinky_y][g.pinky_x] != V)
    {
      muovi(pinky,&g.pinky_x,&g.pinky_y,1);
    }

    if(campo[g.clyde_y][g.clyde_x] != J &&
       campo[g.clyde_y][g.clyde_x] != U &&
       campo[g.clyde_y][g.clyde_x] != V)
    {
      muovi(clyde,&g.clyde_x,&g.clyde_y,1);
    }

  //-verifica imbocco tunnel
    if(g.tuki_x == 23 && g.tuki_y==16)
      {
        g.tuki_x = 3;
        muovi(DESTRA,&g.tuki_x,&g.tuki_y,0);
      }

     if(g.tuki_x == 2 && g.tuki_y==16)
      {
        g.tuki_x = 22;
        muovi(DESTRA,&g.tuki_x,&g.tuki_y,0);
      }

  //-verifica consumo trifogli
  if(campo[g.tuki_y][g.tuki_x] == U)
    {
      mangia();
      campo[g.tuki_y][g.tuki_x] = J;
      copia_campo[g.tuki_y][g.tuki_x] = J;
    }

  //-verifica consumo pillola
   if(campo[g.tuki_y][g.tuki_x] == V)
    {
      mangia();
      campo[g.tuki_y][g.tuki_x] = J;
      copia_campo[g.tuki_y][g.tuki_x] = J;
      superpacman(1);
    }
   else
     {
       if(collisione) return 0;
     }

  return 1;
}

oggetto ** mdl_campo(){

  return copia_campo;
}

posizioni mdl_posizioni()
{
  return g;
}

int mdl_genera_campo(){
  int level=1;
  int l=1,i;
  int h=ALTEZZA;
  int w=LARGHEZZA;

  /** inizializza la posizione dei giocatori */
  g.tuki_y = 25;
  g.tuki_x = 16;
  g.blinky_y = 17;
  g.blinky_x = 14;
  g.inky_y = 17;
  g.inky_x = 14;
  g.pinky_y = 17;
  g.pinky_x = 14;
  g.clyde_y = 17;
  g.clyde_x = 14;

  /** Alloca memoria per il campo */
  copia_campo=(oggetto**)malloc(h*sizeof(oggetto*));
```

```
  if(!copia_campo)return 1;

  for(int in=0;in<h;in++)
    {
      *(copia_campo+in)=malloc(w*sizeof(oggetto));
      if(!(*(copia_campo+in)))return 1;
    }

  /*Copia il campo in campo*/
  for(int i=0;i<h;i+=1)
    {
      for(int j=0;j<w;j+=1)
        {
          copia_campo[i][j]=campo[i][j];
        }
    }

  /*Conto i fiori rimasti nel campo*/
  for(int i=0;i<h;i+=1)
    for(int j=0;j<w;j+=1){
      if(campo[i][j]==TRIFOGLIO) n_trifogli++;
    }

  return 1;
}

void mdl_libera_campo()
{

  for(int in=0;in<ALTEZZA;in++)
    {
      free(*(copia_campo+in));
    }

  free(copia_campo);
}

int mdl_punteggio(){

  return punti;

}
```

Listato tuki_modello.h

```
/*_____________________________________________
* FILE: tuki5_modello.h
*/

/*** SECTION: battlefiled  ***/
#define ALTEZZA 35
#define LARGHEZZA 28

/**
 * Gli oggetti nel campo
 */
typedef enum {
   a='a',
   A = 'A', //ANGOLO ALTO SIN BORDO
   B = 'B', //ANGOLO ALTO DES BORDO
   C = 'C', //ANGOLO BASSO SIN BORDO
   D = 'D', //ANGOLO BASSO DES BORDO
   E = 'E', //MURO ORR BORDO
   F = 'F', //MURO VER BORDO
   G = 'G', //ANGOLO ALTO SIN
   H = 'H', //ANGOLO ALTO DES
   I = 'I', //ANGOLO ALTO DES
   J = 'J', //SPAZIO
   K = 'K',
   L = 'L', //ANGOLO ALTO SIN
   M = 'M', //ANGOLO BASSO DES
   N = 'N', //ANGOLO BASSO SIN
   O = 'O', //ANGOLO BASSO SIN
   P = 'P', //ANGOLO ALTO SIN
   Q = 'Q', //ANGOLO BASSO DES
   R = 'R', //ANGOLO ALTO DES
   S = 'S', //MURO VER
   T = 'T', //MURO ORR
```

```
   U = 'U', //PUNTINO
   V = 'V', //PILLOLA
   X = 'X', //MURO VER BORDO
   Y = 'Y', //ANGOLO BASSO DES
   W = 'W', //ANGOLO BASSO SIN
   Z = 'Z', //MURO ORZZ BORDO
   //Cibo
   TRIFOGLIO ='.',
   PILLOLA = 'O'
} oggetto;

/**
 * I 5 giocatori
 */
typedef enum {
  //Personaggi
  BLINKY='b',
  INKY='i',
  PINKY='p',
  CLYDE='c',
  TUKI='@',

} giocatore;

#ifndef PLAYER
typedef enum  {SINISTRA,SU,DESTRA,GIU,FERMO} direzione;
#define PLAYER
#endif

typedef struct  {
  int tuki_x, tuki_y;
  int blinky_x, blinky_y;
  int inky_x, inky_y;
  int pinky_x, pinky_y;
  int clyde_x, clyde_y;
} posizioni;

/*** interfaccia***/

/* elabora l'evoluzione del gioco e torna 0 quando il tuki è mangiato */
int mdl_passo
(direzione tuki,
 direzione blinky,
 direzione inky,
 direzione pinky,
 direzione clyde);

/* torna una copia del campo ad uso dei giocatori */
oggetto **  mdl_campo();

/* copia delle posizioni dei giocatori, ad uso dei giocatori */
posizioni mdl_posizioni();

/* crea il campo, ad uso del controller */
int mdl_genera_campo();

/* libera la memoria del campo */
void mdl_libera_campo();

/* torna il punteggio del tuki */
int mdl_punteggio();

/* torna lo stato di superpacman */
char mdl_superpacman();
```

Listato tuki_visore.c

```
/*_____________________________________________________+ì
*FILE: tuki5_visore.c
*/

/*** Includes ***/
#include <ctype.h>
#include <locale.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <stdlib.h>
#include <string.h>
#include <termios.h>
```

```
#include <time.h>
#include <unistd.h>
#include "tuki5_modello.h"
#include "tuki5_visore.h"

/*** Define ***/
#define COFFSET(width) (cols-width)/2
#define ROFFSET(heigh) (rows-heigh)/2

/*** Data ***/
struct termios orig_termios;
int rows,cols;

/*** Terminal ***/
void die(const char *s) {
  write(STDOUT_FILENO, "\x1b[2J", 4);
  write(STDOUT_FILENO, "\x1b[H", 3);

  perror(s);
  exit(1);
}

int leggi_dimensioni_terminale(int *rows, int *cols) {
  struct winsize ws;
  if (ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws) == -1 ||
      ws.ws_col == 0)
    {
    return -1;
  } else {
    *cols = ws.ws_col;
    *rows = ws.ws_row;
    return 0;
  }
}

void disabilita_modo_raw()
{
  if (
      tcsetattr
      (STDIN_FILENO, TCSAFLUSH, &orig_termios) == -1)
    die("tcsetattr");
}

void abilita_modo_raw() {

  if (tcgetattr(STDIN_FILENO, &orig_termios) == -1)
    die("tcgetattr");
  atexit(disabilita_modo_raw);

  struct termios raw = orig_termios;
  raw.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);
  raw.c_oflag &= ~(OPOST);
  raw.c_cflag |= (CS8);
  raw.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);
  raw.c_cc[VMIN] = 0;
  raw.c_cc[VTIME] = 1;
  if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw) == -1) die("tcsetattr");
}

void view_init()
{
  //abilita_modo_raw();
  leggi_dimensioni_terminale(&rows, &cols);
  setlocale(LC_CTYPE, "");
}

/*** output ***/
char * terminale_str(oggetto e)
{
  static char bc,ic,pc,cc,tc=0;
  static char st[2];
  st[0]=e;
  st[1]=0;
  bc=ic=pc=cc=tc=0;
  if(e==BLINKY)
    {
      if(bc==0)
        {
          bc=1;
          return "\x1b[1mb";
          //return "\u2229";
        }
      else
```

```
      {
        bc=0;
        return "B";
        //return "\u2126";
      }
  }

 if(e==INKY)
  {
    if(ic==0)
      {
        ic=1;
        return "\x1b[1mi";
      }
    else
      {
        ic=0;
        return "i";
      }
  }

  if(e==PINKY)
  {
    if(pc==0)
      {
        pc=1;
        return "\x1b[1mp";
      }
    else
      {
        pc=0;
        return "_";
      }
  }

   if(e==CLYDE)
  {
    if(cc==0)
      {
        cc=1;
        return "\x1b[1mc";
      }
    else
      {
        cc=0;
        return "c";
      }
  }

   if(e==TUKI)
  {
    if(tc==0)
      {
        tc=1;
        return "\x1b[1m@";
      }
    else
      {
        tc=0;
        return "O";
      }
  }

if(e==B)
  return "\u2557";

if(e==A)
  return "\x1b[38:5:21m\u2554";

if(e==J)
  return " ";

if(e==C)
  return "\u255a";

if(e==D)
  return "\u255D";

if(e==E)
  return "\u2550";

if(e==F)
  return "\u2551";
```

```
  if(e==G)
    return "\u250c";

  if(e==H)
    return "\u2510";

  if(e==K)
    return "\u2596";

  if(e==I)
    return "\u2555";

  if(e==L)
    return "\u2552";

  if(e==M)
     return "\u259F";

   if(e==N)
     return "\u2599";

   if(e==O)
     return "\u2559";

   if(e==P)
     return "\u2553";

   if(e==Q)
     return "\u255c";

   if(e==R)
     return "\u2556";

   if(e==S)
     return "\u2502";

   if(e==T)
      return "\u2500";

   if(e==W)
     return "\u2514";

   if(e==X)
     return "\u2551";

     if(e==Y)
       return "\u2518";

     if(e==V) //PIllola
       return "\x1b[38:5:231m\u2022\x1b[38:5:21m";

     if(e==U) //Trifogli
       return "\x1b[38:5:231m.\x1b[38:5:21m";
     //return "\x1b[38:5:231m\u2022\x1b[38:5:21m";

     if(e==Z)
       return "\u2550";

     if(e==a)
       return "\u2598";

  return st;
}

void view_labirinto(oggetto **campo)
{
  int x,y;
  int rof=ROFFSET(ALTEZZA*1);
  int cof=COFFSET(LARGHEZZA);
  char str[25];

  for(int i=0;i<ALTEZZA;i++)
    for(int j=0;j<LARGHEZZA;j++)
      {
        x=j+cof;
        y=i+rof;
        sprintf(str,"\x1b[%d;%dH%s",y,x,terminale_str(*(*(campo+i)+j)));
        write(STDOUT_FILENO, str, strlen(str));
      }
  printf("\x1b[%d;%dH",y,1);
}
```

```
int view_sfondo(int r, int c,oggetto **campo)
{
  int x,y;
  int rof=ROFFSET(ALTEZZA*1);
  int cof=COFFSET(LARGHEZZA);
  char str[25];

  x=c+cof;
  y=r+rof;
  sprintf(str,"\x1b[%d;%dH%s",y,x,terminale_str(*(*(campo+r)+c)));
  write(STDOUT_FILENO, str, strlen(str));

  return 0;
}

void view_mangiato(posizioni g)
{
  int x,y;
  int rof=ROFFSET(ALTEZZA);
  int cof=COFFSET(LARGHEZZA);
  char str[80];
  char* str_lst="@*[]\\||//#+^@";

  x=g.tuki_x+cof;
  y=g.tuki_y+rof;
  for(int i=0;i<strlen(str_lst);i++)
    {
      sprintf(str,"\x1b[38:5:229m\x1b[%d;%dH%c",y,x,*(str_lst+i));
      write(STDOUT_FILENO, str, strlen(str));
      delay(50);
    }
}

int view_giocatori(posizioni g, oggetto ** lab,char in_blue)
{
  static posizioni g_prec;
  static char init=0;
  char pu[20];
  /* colori dei fantasmi */
  int bl,in,pi,cl;
  if(in_blue)
    {
      strcpy(pu,"\x1b[5m");
      bl=in=pi=cl=57;
    }
  else
    {
      strcpy(pu,"");
      bl=196;
      in=117;
      pi=218;
      cl=215;
    }

  if(init==0)
    {
      g_prec=g;
      init=1;
    }

  int x,y;
  int rof=ROFFSET(ALTEZZA);
  int cof=COFFSET(LARGHEZZA);
  char str[80];

  /** Tuki */
  // Ripristina il campo
  x = g_prec.tuki_x;
  y = g_prec.tuki_y;
  sprintf(str,"\x1b[38:5:229m\x1b[%d;%dH%s",
          y+rof,x+cof,terminale_str(*(*(lab+y)+x)));
  write(STDOUT_FILENO, str, strlen(str));

  //Stampa Tuki
  x=g.tuki_x+cof;
  y=g.tuki_y+rof;
  sprintf(str,"\x1b[38:5:229m\x1b[%d;%dH%s",
          y,x,terminale_str(TUKI));
  write(STDOUT_FILENO, str, strlen(str));

  /** Blinky */
  // Ripristina il campo
  x = g_prec.blinky_x;
```

```
  y = g_prec.blinky_y;
  sprintf(str,"\x1b[38:5:229m\x1b[%d;%dH%s",
          y+rof,x+cof,terminale_str(*(*(lab+y)+x)));
  write(STDOUT_FILENO, str, strlen(str));

  // stampa Blinky
  x=g.blinky_x+cof;
  y=g.blinky_y+rof;
  sprintf(str,"%s\x1b[38:5:%dm\x1b[%d;%dH%s\x1b[0m",
          pu,bl,y,x,terminale_str(BLINKY));
  write(STDOUT_FILENO, str, strlen(str));

  /** Inky */
   // Ripristina il campo
  x = g_prec.inky_x;
  y = g_prec.inky_y;
  sprintf(str,"\x1b[38:5:229m\x1b[%d;%dH%s",
          y+rof,x+cof,terminale_str(*(*(lab+y)+x)));
  write(STDOUT_FILENO, str, strlen(str));

  // stampa inky
  x=g.inky_x+cof;
  y=g.inky_y+rof;
  sprintf(str,"%s\x1b[38:5:%dm\x1b[%d;%dH%s\x1b[0m",
          pu,in,y,x,terminale_str(INKY));
  write(STDOUT_FILENO, str, strlen(str));

  /** Pinky */
  // Ripristina il campo
  x = g_prec.pinky_x;
  y = g_prec.pinky_y;
  sprintf(str,"\x1b[38:5:229m\x1b[%d;%dH%s",
          y+rof,x+cof,terminale_str(*(*(lab+y)+x)));
  write(STDOUT_FILENO, str, strlen(str));

  //stampa pinky
  x=g.pinky_x+cof;
  y=g.pinky_y+rof;
  sprintf(str,"%s\x1b[38:5:%dm\x1b[%d;%dH%s\x1b[0m",
          pu,pi,y,x,terminale_str(PINKY));
  write(STDOUT_FILENO, str, strlen(str));

  /** Clyde */
  // Ripristina il campo
  x = g_prec.clyde_x;
  y = g_prec.clyde_y;
  sprintf(str,"\x1b[38:5:229m\x1b[%d;%dH%s",
          y+rof,x+cof,terminale_str(*(*(lab+y)+x)));
  write(STDOUT_FILENO, str, strlen(str));

  // stampa clyde
  x=g.clyde_x+cof;
  y=g.clyde_y+rof;
  sprintf(str,"%s\x1b[38:5:%dm\x1b[%d;%dH%s\x1b[0m",
          pu,cl,y,x,terminale_str(CLYDE));
  write(STDOUT_FILENO, str, strlen(str));

  /** Memorizza le posizioni del turno appena visualizzato */
  g_prec=g;
}

void view_punteggio(int punti)
{
  char str2[50];
  char str[60];
  sprintf(str2,"Score %d ",punti);
  int l=strlen(str2);
  int rof=ROFFSET(ALTEZZA*1);
  int cof=COFFSET(l);
  sprintf(str,"\x1b[%d;%dH%s",ALTEZZA+2,cof,str2);
  write(STDOUT_FILENO, str, strlen(str));
}

void view_gameover(char * message)
{
  char str2[50];
  char str[60];

  int l=strlen(message);
  int rof=ROFFSET(ALTEZZA*1);
  int cof=COFFSET(l);
  sprintf(str,"\x1b[%d;%dH%s\r\n\r\n",rof+12+1,cof,message);
  write(STDOUT_FILENO, str, strlen(str));
```

```
  //Set cursor at the begin of last line
   printf("\x1b[%d;%dH",rows,1);
}

void delay(int milliseconds){
  long pause;
  clock_t now,then;

  pause = milliseconds*(CLOCKS_PER_SEC/1000);
  now = then = clock();
  char c;
  while( (now-then) < pause )
    {
      now = clock();
    }
}

void fading( char * str,int r, int c)
{
  float gray=232;
  float inc=0.03;
  float vrs=1;
  while(gray<255)
    {
      printf("\x1b[%d;%dH\x1b[38;5;%dm%s\x1b[m",r,c,(int)gray,str);
      if(gray<255){
        vrs=+1;
      }
      if(gray<=232){
        //vrs=1;
      }
      delay(1);
      gray+=vrs*inc;
    }
  fflush(stdout);
}

void view_presentazione(){
  char str2[50];
  char str[60];
  char * message[]=
    {"TUKI e Pacman",
     "Coding-game prodotto e distribuito",
     "da Scuola Sisini",
     "visita http://pumar.it"};
  int l,rof,cof;

  printf("\x1b[%d;%dH",1,1);
  write(STDOUT_FILENO, "\x1b[2J", 4);
  for(int i=0;i<4;i++)
    {
      l=strlen(message[i]);
      rof=ROFFSET(ALTEZZA*1.5);
      cof=COFFSET(l);
      fading(message[i],rof+ALTEZZA/2+i,cof);
      delay(100);
    }
  printf("\x1b[%d;%dH",1,1);
  delay(1250);
  write(STDOUT_FILENO, "\x1b[2J", 4);

}
```

Listato tuki_visore.h

```
/*_____________________________________________________________
* FILE: tuki5_visore.h
*/

void view_init();

/** Stampa il labirinto */
void view_labirinto(oggetto **labirinto);

/** stampa l'oggetto (se) presente in posizione r e c */
int view_sfondo(int r, int c,oggetto **labirinto);

/** stampa i giocatori sul campo */
int view_giocatori
```

```
(posizioni giocatori, oggetto ** labirinto, char inblue);

/** stampa il punteggio */
void view_punteggio(int punti);

/** stampa la fine del gioco */
void view_gameover(char * messaggio);

/** stamp la presentazione */
void view_presentazione();

/** stampa la fine di pacman */
void view_mangiato(posizioni g);

void delay(int millis);
```

Listato gioca_fantasmi.c

```
/*_____________________________________________________________
* FILE: gioca_fantasmi.c
*/

#include "tuki5_modello.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/*_____________________________________________________________
 *|
 *| Nodo del grafo che rappresenta il labirinto
 */
typedef struct {
  int riga;
  int colonna;
  int indice;
  int n_sx;
  int n_dx;
  int n_su;
  int n_giu;
} nodo;

/*_____________________________________________________________
 *|
 *| I nodi del grafo el labirinto
 */
nodo grafo[34];

/*_____________________________________________________________
 *|
 *| Crea un grafo corrispondente al labirinto del tuki
 */
void collega_nodi()
{
  grafo[0].riga = 3;
  grafo[0].colonna = 6;
  grafo[0].indice = 0;
  grafo[0].n_sx = 2;
  grafo[0].n_dx = 5;
  grafo[0].n_su = -1;
  grafo[0].n_giu = 3;

  grafo[1].riga = 3;
  grafo[1].colonna = 21;
  grafo[1].indice = 1;
  grafo[1].n_sx = 6;
  grafo[1].n_dx = 9;
  grafo[1].n_su = -1;
  grafo[1].n_giu = 8;

  grafo[2].riga = 7;
  grafo[2].colonna = 1;
  grafo[2].indice = 2;
  grafo[2].n_sx = -1;
  grafo[2].n_dx = 3;
  grafo[2].n_su = 0;
  grafo[2].n_giu = 10;

  grafo[3].riga = 7;
  grafo[3].colonna = 6;
```

```
grafo[3].indice = 3;
grafo[3].n_sx = 2;
grafo[3].n_dx = 4;
grafo[3].n_su = 0;
grafo[3].n_giu = 10;

grafo[4].riga = 7;
grafo[4].colonna = 9;
grafo[4].indice = 4;
grafo[4].n_sx = 3;
grafo[4].n_dx = 5;
grafo[4].n_su = -1;
grafo[4].n_giu = 12;

grafo[5].riga = 7;
grafo[5].colonna = 12;
grafo[5].indice = 5;
grafo[5].n_sx = 4;
grafo[5].n_dx = 6;
grafo[5].n_su = 0;
grafo[5].n_giu = -1;

grafo[6].riga = 7;
grafo[6].colonna = 15;
grafo[6].indice = 6;
grafo[6].n_sx = 5;
grafo[6].n_dx = 7;
grafo[6].n_su = 1;
grafo[6].n_giu = -1;

grafo[7].riga = 7;
grafo[7].colonna = 18;
grafo[7].indice = 7;
grafo[7].n_sx = 6;
grafo[7].n_dx = 8;
grafo[7].n_su = -1;
grafo[7].n_giu = 13;

grafo[8].riga = 7;
grafo[8].colonna = 21;
grafo[8].indice = 8;
grafo[8].n_sx = 7;
grafo[8].n_dx = 9;
grafo[8].n_su = 1;
grafo[8].n_giu = 11;

grafo[9].riga = 7;
grafo[9].colonna = 26;
grafo[9].indice = 9;
grafo[9].n_sx = 8;
grafo[9].n_dx = -1;
grafo[9].n_su = 1;
grafo[9].n_giu = 11;

grafo[10].riga = 10;
grafo[10].colonna = 6;
grafo[10].indice = 10;
grafo[10].n_sx = 2;
grafo[10].n_dx = -1;
grafo[10].n_su = 3;
grafo[10].n_giu = 14;

grafo[11].riga = 10;
grafo[11].colonna = 21;
grafo[11].indice = 11;
grafo[11].n_sx = -1;
grafo[11].n_dx = 9;
grafo[11].n_su = 8;
grafo[11].n_giu = 17;

grafo[12].riga = 13;
grafo[12].colonna = 12;
grafo[12].indice = 12;
grafo[12].n_sx = 15;
grafo[12].n_dx = 13;
grafo[12].n_su = 4;
grafo[12].n_giu = -1;

grafo[13].riga = 13;
grafo[13].colonna = 15;
grafo[13].indice = 13;
grafo[13].n_sx = 12;
grafo[13].n_dx = 16;
```

```
grafo[13].n_su = 7;
grafo[13].n_giu = -1;

grafo[14].riga = 16;
grafo[14].colonna = 6;
grafo[14].indice = 14;
grafo[14].n_sx = -1;
grafo[14].n_dx = 15;
grafo[14].n_su = 10;
grafo[14].n_giu = 20;

grafo[15].riga = 16;
grafo[15].colonna = 9;
grafo[15].indice = 15;
grafo[15].n_sx = 14;
grafo[15].n_dx = -1;
grafo[15].n_su = 12;
grafo[15].n_giu = 18;

grafo[16].riga = 16;
grafo[16].colonna = 18;
grafo[16].indice = 16;
grafo[16].n_sx = -1;
grafo[16].n_dx = 17;
grafo[16].n_su = 13;
grafo[16].n_giu = 19;

grafo[17].riga = 16;
grafo[17].colonna = 21;
grafo[17].indice = 17;
grafo[17].n_sx = 16;
grafo[17].n_dx = -1;
grafo[17].n_su = 11;
grafo[17].n_giu = 23;

grafo[18].riga = 19;
grafo[18].colonna = 9;
grafo[18].indice = 18;
grafo[18].n_sx = -1;
grafo[18].n_dx = 19;
grafo[18].n_su = 15;
grafo[18].n_giu = 21;

grafo[19].riga = 19;
grafo[19].colonna = 18;
grafo[19].indice = 19;
grafo[19].n_sx = 18;
grafo[19].n_dx = -1;
grafo[19].n_su = 16;
grafo[19].n_giu = 22;

grafo[20].riga = 22;
grafo[20].colonna = 6;
grafo[20].indice = 20;
grafo[20].n_sx = 30;
grafo[20].n_dx = 21;
grafo[20].n_su = 14;
grafo[20].n_giu = 24;

grafo[21].riga = 22;
grafo[21].colonna = 9;
grafo[21].indice = 21;
grafo[21].n_sx = 20;
grafo[21].n_dx = 26;
grafo[21].n_su = 18;
grafo[21].n_giu = -1;

grafo[22].riga = 22;
grafo[22].colonna = 18;
grafo[22].indice = 22;
grafo[22].n_sx = 27;
grafo[22].n_dx = 23;
grafo[22].n_su = 19;
grafo[22].n_giu = -1;

grafo[23].riga = 22;
grafo[23].colonna = 21;
grafo[23].indice = 23;
grafo[23].n_sx = 22;
grafo[23].n_dx = 31;
grafo[23].n_su = 17;
grafo[23].n_giu = 29;
```

```
  grafo[24].riga = 25;
  grafo[24].colonna = 6;
  grafo[24].indice = 24;
  grafo[24].n_sx = -1;
  grafo[24].n_dx = 25;
  grafo[24].n_su = 20;
  grafo[24].n_giu = 30;

  grafo[25].riga = 25;
  grafo[25].colonna = 9;
  grafo[25].indice = 25;
  grafo[25].n_sx = 24;
  grafo[25].n_dx = 26;
  grafo[25].n_su = -1;
  grafo[25].n_giu = 32;

  grafo[26].riga = 25;
  grafo[26].colonna = 12;
  grafo[26].indice = 26;
  grafo[26].n_sx = 25;
  grafo[26].n_dx = 27;
  grafo[26].n_su = 21;
  grafo[26].n_giu = -1;

  grafo[27].riga = 25;
  grafo[27].colonna = 15;
  grafo[27].indice = 27;
  grafo[27].n_sx = 26;
  grafo[27].n_dx = 28;
  grafo[27].n_su = 22;
  grafo[27].n_giu = -1;

  grafo[28].riga = 25;
  grafo[28].colonna = 18;
  grafo[28].indice = 28;
  grafo[28].n_sx = 27;
  grafo[28].n_dx = 29;
  grafo[28].n_su = -1;
  grafo[28].n_giu = 33;

  grafo[29].riga = 25;
  grafo[29].colonna = 21;
  grafo[29].indice = 29;
  grafo[29].n_sx = 28;
  grafo[29].n_dx = -1;
  grafo[29].n_su = 23;
  grafo[29].n_giu = 31;

  grafo[30].riga = 28;
  grafo[30].colonna = 3;
  grafo[30].indice = 30;
  grafo[30].n_sx = 32;
  grafo[30].n_dx = 24;
  grafo[30].n_su = 20;
  grafo[30].n_giu = -1;

  grafo[31].riga = 28;
  grafo[31].colonna = 24;
  grafo[31].indice = 31;
  grafo[31].n_sx = 29;
  grafo[31].n_dx = 33;
  grafo[31].n_su = 23;
  grafo[31].n_giu = -1;

  grafo[32].riga = 31;
  grafo[32].colonna = 12;
  grafo[32].indice = 32;
  grafo[32].n_sx = 30;
  grafo[32].n_dx = 33;
  grafo[32].n_su = 25;
  grafo[32].n_giu = -1;

  grafo[33].riga = 31;
  grafo[33].colonna = 15;
  grafo[33].indice = 33;
  grafo[33].n_sx = 32;
  grafo[33].n_dx = 31;
  grafo[33].n_su = 28;
  grafo[33].n_giu = -1;

}

/*_____________________________________________________________
```

```
 *|
 *| Torna l'indice del nodo presente in (riga, colonna)
 *| Se non è presente alcun nodo (il fantasma è su un ramo)
 *| torna -1
 */
int indice_nodo_cella(int riga, int col)
{
  for(int i=0; i<34;i++)
    {
      if(grafo[i].colonna == col && grafo[i].riga == riga)
        return grafo[i].indice;
    }
  return -1;
}

/*_____________________________________________________________
 *|
 *| Una cella del labirinto
 */
typedef struct {
  int riga;
  int colonna;
} cella;

/*_____________________________________________________________
 *|
 *| restituisce la direzione in cui si deve muovere il fantasma
 *| sceliendo tra le due libere quella diversa dalla
 *| cella <precedete>
 *| Il comportamento è deterministico solo se la cella
 *| corrente non è un nodo del grafo
 */
direzione prossima_cella
(cella corrente, cella precedente, oggetto **lab)
{
  int riga = corrente.riga;
  int colonna = corrente.colonna;

 /* oggetti ai 4 angoli della cella corrente */
  oggetto o_sx, o_dx, o_su, o_giu;
  o_sx = lab[riga][colonna-1];
  o_dx = lab[riga][colonna+1];
  o_su = lab[riga-1][colonna];
  o_giu = lab[riga+1][colonna];

  if(o_sx == J || o_sx == U  || o_sx == V )
    {
      if(precedente.colonna != colonna - 1)
          return SINISTRA;
    }

   if(o_dx == J || o_dx == U || o_dx == V)
    {
      if(precedente.colonna != colonna + 1)
        return DESTRA;
    }

    if(o_su == J || o_su == U || o_su == V)
    {
      if(precedente.riga != riga - 1)
        return SU;
    }

    if(o_giu == J || o_giu == U || o_giu == V)
      {
        if(precedente.riga != riga + 1)
          return GIU;
      }
}

/*_____________________________________________
 *|
 *| Distanza  tra due celle
 */
double distanza_celle(cella a, cella b)
{
  double d =
    (a.riga-b.riga)*(a.riga-b.riga)+
    (a.colonna-b.colonna)*(a.colonna-b.colonna);
  return sqrt(d);
}

direzione gioca_blinky(posizioni p, oggetto **lab)
```

```
{
  static int turno = 0;

  static int colonna_old,riga_old;

  turno+=1;

  /* indica l'edge su cui si sta muovendo il fantasma */
  static nodo nodo_da, nodo_a;

  int riga = p.blinky_y;
  int colonna = p.blinky_x;

  direzione ld =  FERMO;

  if(turno == 1)
    {
      collega_nodi();
    }

  /* uscita dalla casetta dei fantasmi */
  if(turno<7)
    {
      if(turno<5)
        return SU;

      nodo_a = grafo[12];
      nodo_da = grafo[12];

      colonna_old = -1;
      colonna = nodo_a.colonna;

      riga_old = -1;
      riga = nodo_a.riga;
      return SINISTRA;
    }

  /* Sono su un nodo? */
  int inx = indice_nodo_cella(riga,colonna);

  /* cella su cui si trova e cella precedente */
  cella c, pr;
  c.riga = riga;
  c.colonna = colonna;
  pr.riga = riga_old;
  pr.colonna = colonna_old;

  if(inx<0)
    {
      /*_____________________________________________________
       *|  Non sono su un nodo. Il percorso è obbligato
       *| sono definiti: nodo_da, nodo_a, x e y _old
       *| satbilisce il prossimo passo come l'unico
       *| possibile in modo che non sia ne un muro ne la
       *| posizione attuale
       */

      ld = prossima_cella(c, pr, lab);

      colonna_old = colonna;
      riga_old = riga;

      return ld;
    }
  else
    {
      /*_____________________________________________________
       *|
       *| Sono su un nodo, devo decidere il prossimo
       *| edge su cui muovermi
       *| se non ci sono stati errori inx ==  nodo_a.indice
       *| in base alla posizione di tuki.
       *| Al termine di questo blocco devo:
       *| nodo_da <- noda_a
       *| nodo_a <- nodo_dest
       */

      if(inx != nodo_a.indice)
        {
          //exit(0);
        }

      /* Nodo temporaneo */
```

```
nodo n_d;

/* calcola la distanza tra i prossimi nodi e tuki */
double d_sx = 10000, d_dx = 10000, d_su = 10000, d_giu = 10000;

cella tuki;
tuki.riga = p.tuki_y;
tuki.colonna = p.tuki_x;

cella s;

if(nodo_a.n_sx >= 0)
  {
    n_d = grafo[nodo_a.n_sx];
    if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
      {
        s.riga = n_d.riga;
        s.colonna = n_d.colonna;
        d_sx = distanza_celle(s, tuki);
      }
  }
if(nodo_a.n_dx >= 0)
  {
    n_d = grafo[nodo_a.n_dx];
    if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
      {
        s.riga = n_d.riga;
        s.colonna = n_d.colonna;
        d_dx = distanza_celle(s, tuki);
      }
  }
if(nodo_a.n_su >= 0)
  {
    n_d = grafo[nodo_a.n_su];
    if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
      {
        s.riga = n_d.riga;
        s.colonna = n_d.colonna;
        d_su = distanza_celle(s, tuki);
      }
  }
if(nodo_a.n_giu >= 0)
  {
    n_d = grafo[nodo_a.n_giu];
    if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
      {
        s.riga = n_d.riga;
        s.colonna = n_d.colonna;
        d_giu = distanza_celle(s, tuki);
      }
  }
    nodo_da =  nodo_a;

int ind;
/* cerco il nodo più vicino a tuki */
if(d_sx<=d_dx && d_sx<=d_su && d_sx<=d_giu)
  {
    ind = nodo_a.n_sx;
    nodo_a = grafo[ind];

    ld = SINISTRA;
  }else if(d_dx<=d_sx && d_dx<=d_su && d_dx<=d_giu)
  {
    ind = nodo_a.n_dx;
    nodo_a = grafo[ind];

    ld = DESTRA;
  } else if(d_su<=d_sx && d_su<=d_dx && d_su<=d_giu)
  {
    ind = nodo_a.n_su;
    nodo_a = grafo[ind];

    ld = SU;
  } else if(d_giu<=d_sx && d_giu<=d_dx && d_giu<=d_su)
  {
    ind = nodo_a.n_giu;

    nodo_a = grafo[ind];

    ld = GIU;
  }

colonna_old = colonna;
```

```
      riga_old = riga;

      if(nodo_da.indice == nodo_a.indice)
        {
          //exit(0);
        }

      return ld;

    }

  return ld;
}

direzione gioca_pinky(posizioni p, oggetto **lab)
{

  static int turno = 0;

  static int colonna_old,riga_old;

  static cella tuki, tuki_old;

  turno+=1;

  /* indica l'edge su cui si sta muovendo il fantasma */
  static nodo nodo_da, nodo_a;

  int riga = p.pinky_y;
  int colonna = p.pinky_x;

  direzione ld =  FERMO;

  /* uscita dalla casetta dei fantasmi */
  if(turno<5)
    return FERMO;
  if(turno<12 && turno>5)
    {
      if(turno<10)
        return SU;

      nodo_a = grafo[13];
      nodo_da = grafo[13];

      colonna_old = -1;
      colonna = nodo_a.colonna;

      riga_old = -1;
      riga = nodo_a.riga;
      return DESTRA;
    }

  /* Sono su un nodo? */
  int inx = indice_nodo_cella(riga,colonna);

  /* cella su cui si trova e cella precedente */
  cella c, pr;
  c.riga = riga;
  c.colonna = colonna;
  pr.riga = riga_old;
  pr.colonna = colonna_old;

  if(inx<0)
    {
      /*_____________________________________________
       *|  Non sono su un nodo. Il percorso è obbligato
       *| sono definiti: nodo_da, nodo_a, x e y _old
       *| satbilisce il prossimo passo come l'unico
       *| possibile in modo che non sia ne un muro ne la
       *| posizione attuale
       */

      ld = prossima_cella(c, pr, lab);

      colonna_old = colonna;
      riga_old = riga;

      return ld;
    }
  else
    {
      /*_____________________________________________
       *|
```

```
 *| Sono su un nodo, devo decidere il prossimo
 *| edge su cui muovermi
 *| se non ci sono stati errori inx ==  nodo_a.indice
 *| in base alla posizione di tuki.
 *| Al termine di questo blocco devo:
 *| nodo_da <- noda_a
 *| nodo_a <- nodo_dest
 */

if(inx != nodo_a.indice)
  {
    //exit(0);
  }

/* Nodo temporaneo */
nodo n_d;

/* calcola la distanza tra i prossimi nodi e la cella 4 passi
 avanti a Tuki */
double d_sx = 10000, d_dx = 10000, d_su = 10000, d_giu = 10000;

cella s;
tuki.riga = p.tuki_y;
tuki.colonna = p.tuki_x;

/* In che direzione muove Tuki?
 * righe o colonne?
 * crescenti o descresceti?
 */
int d_colonna, d_riga;
d_colonna = tuki.colonna - tuki_old.colonna;
d_riga = tuki.riga - tuki_old.riga;
tuki_old.colonna = tuki.colonna;
tuki_old.riga = tuki.riga;

/* stabilisco la cella 4 passi avanti a Tuki */
if(d_colonna !=0)
  {
    tuki.colonna += 4*d_colonna;
  }
else if(d_riga != 0)
  {
    tuki.riga += 4*d_riga;
  }
else
  {
    tuki.riga = 7;
    tuki.colonna= 6;
  }

if(nodo_a.n_sx >= 0)
  {
    n_d = grafo[nodo_a.n_sx];
    if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
      {
        s.riga = n_d.riga;
        s.colonna = n_d.colonna;
        d_sx = distanza_celle(s, tuki);
      }
  }

if(nodo_a.n_dx >= 0)
  {
    n_d = grafo[nodo_a.n_dx];
    if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
      {
        s.riga = n_d.riga;
        s.colonna = n_d.colonna;
        d_dx = distanza_celle(s, tuki);
      }
  }

if(nodo_a.n_su >= 0)
  {
    n_d = grafo[nodo_a.n_su];
    if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
      {
        s.riga = n_d.riga;
        s.colonna = n_d.colonna;
        d_su = distanza_celle(s, tuki);
      }
  }
```

```
      if(nodo_a.n_giu >= 0)
        {
          n_d = grafo[nodo_a.n_giu];
          if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
            {
              s.riga = n_d.riga;
              s.colonna = n_d.colonna;
              d_giu = distanza_celle(s, tuki);
            }
        }

      nodo_da =  nodo_a;

      int ind;
      /* cerco il nodo più vicino a tuki */
      if(d_sx<=d_dx && d_sx<=d_su && d_sx<=d_giu)
        {
          ind = nodo_a.n_sx;
          nodo_a = grafo[ind];

          ld = SINISTRA;
        }else if(d_dx<=d_sx && d_dx<=d_su && d_dx<=d_giu)
        {
          ind = nodo_a.n_dx;
          nodo_a = grafo[ind];

          ld = DESTRA;
        } else if(d_su<=d_sx && d_su<=d_dx && d_su<=d_giu)
        {
          ind = nodo_a.n_su;
          nodo_a = grafo[ind];

          ld = SU;
        } else if(d_giu<=d_sx && d_giu<=d_dx && d_giu<=d_su)
        {
          ind = nodo_a.n_giu;

          nodo_a = grafo[ind];

          ld = GIU;
        }

      colonna_old = colonna;
      riga_old = riga;

      if(nodo_da.indice == nodo_a.indice)
        {
          printf("Errore: da %d != a %d direzione: %d",
                 nodo_da.indice,nodo_a.indice,ld);
          //exit(0);
        }

      return ld;

    }
  return ld;
}

direzione gioca_inky(posizioni p, oggetto **lab)
{
  static int turno = 0;
  static int colonna_old,riga_old;

  turno+=1;

  /* indica l'edge su cui si sta muovendo il fantasma */
  static nodo nodo_da, nodo_a;

  int riga = p.inky_y;
  int colonna = p.inky_x;

  direzione ld =  FERMO;

  /* uscita dalla casetta dei fantasmi */
  if(turno<10)
    return FERMO;
  if(turno<17 && turno>10)
    {
      if(turno<15)
        return SU;

      nodo_a = grafo[13];
      nodo_da = grafo[13];
```

```
    colonna_old = -1;
    colonna = nodo_a.colonna;

    riga_old = -1;
    riga = nodo_a.riga;
    return DESTRA;
  }

/* Sono su un nodo? */
int inx = indice_nodo_cella(riga,colonna);

/* cella su cui si trova e cella precedente */
cella c, pr;
c.riga = riga;
c.colonna = colonna;
pr.riga = riga_old;
pr.colonna = colonna_old;

if(inx<0)
  {
    /*_____________________________________________
     *|  Non sono su un nodo. Il percorso è obbligato
     *| sono definiti: nodo_da, nodo_a, x e y _old
     *| satbilisce il prossimo passo come l'unico
     *| possibile in modo che non sia ne un muro ne la
     *| posizione attuale
     */

    ld = prossima_cella(c, pr, lab);

    colonna_old = colonna;
    riga_old = riga;

    return ld;
  }
else
  {
    /*_____________________________________________
     *|
     *| Sono su un nodo, devo decidere il prossimo
     *| edge su cui muovermi
     *| se non ci sono stati errori inx ==  nodo_a.indice
     *| in base alla posizione di tuki.
     *| Al termine di questo blocco devo:
     *| nodo_da <- noda_a
     *| nodo_a <- nodo_dest
     */

    if(inx != nodo_a.indice)
      {
        //exit(0);
      }

    /* Nodo temporaneo */
    nodo n_d;

    /* calcola la distanza tra i prossimi nodi e la cella intermedia
       tra Tuki e Blinky */
    double d_sx = 10000, d_dx = 10000, d_su = 10000, d_giu = 10000;

    cella s;
    cella tuki, blinky;
    tuki.riga = p.tuki_y;
    tuki.colonna = p.tuki_x;
    blinky.riga = p.blinky_y;
    blinky.colonna = p.blinky_x;

    int m_colonna, m_riga;
    m_colonna = (tuki.colonna + blinky.colonna)/2;
    m_riga = (tuki.riga + blinky.riga)/2;

    /* stabilisco la cella tra Tuki e Blinky */

    tuki.riga = m_riga;
    tuki.colonna = m_colonna;

    if(nodo_a.n_sx >= 0)
      {
        n_d = grafo[nodo_a.n_sx];
        if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
          {
            s.riga = n_d.riga;
```

```
            s.colonna = n_d.colonna;
            d_sx = distanza_celle(s, tuki);
          }
      }

    if(nodo_a.n_dx >= 0)
      {
        n_d = grafo[nodo_a.n_dx];
        if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
          {
            s.riga = n_d.riga;
            s.colonna = n_d.colonna;
            d_dx = distanza_celle(s, tuki);
          }
      }

    if(nodo_a.n_su >= 0)
      {
        n_d = grafo[nodo_a.n_su];
        if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
          {
            s.riga = n_d.riga;
            s.colonna = n_d.colonna;
            d_su = distanza_celle(s, tuki);
          }
      }

    if(nodo_a.n_giu >= 0)
      {
        n_d = grafo[nodo_a.n_giu];
        if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
          {
            s.riga = n_d.riga;
            s.colonna = n_d.colonna;
            d_giu = distanza_celle(s, tuki);
          }
      }

    nodo_da =  nodo_a;

    int ind;
    /* cerco il nodo più vicino a tuki */
    if(d_sx<=d_dx && d_sx<=d_su && d_sx<=d_giu)
      {
        ind = nodo_a.n_sx;
        nodo_a = grafo[ind];

        ld = SINISTRA;
      }else if(d_dx<=d_sx && d_dx<=d_su && d_dx<=d_giu)
      {
        ind = nodo_a.n_dx;
        nodo_a = grafo[ind];

        ld = DESTRA;
      } else if(d_su<=d_sx && d_su<=d_dx && d_su<=d_giu)
      {
        ind = nodo_a.n_su;
        nodo_a = grafo[ind];

        ld = SU;
      } else if(d_giu<=d_sx && d_giu<=d_dx && d_giu<=d_su)
      {
        ind = nodo_a.n_giu;

        nodo_a = grafo[ind];

        ld = GIU;
      }

    colonna_old = colonna;
    riga_old = riga;

    if(nodo_da.indice == nodo_a.indice)
      {
        printf("Errore: da %d != a %d direzione: %d",
               nodo_da.indice,nodo_a.indice,ld);
        //exit(0);
      }

    return ld;
  }
 return ld;
}
```

```
direzione gioca_clyde(posizioni p, oggetto **lab)
{
  static int turno = 0;

  static int colonna_old,riga_old;

  turno+=1;

  /* indica l'edge su cui si sta muovendo il fantasma */
  static nodo nodo_da, nodo_a;

  int riga = p.clyde_y;
  int colonna = p.clyde_x;

  direzione ld =  FERMO;
  static FILE * f;
  if(turno == 1)
    {
      collega_nodi();
      //f=fopen("debugs.txt","w+");
    }

  /* uscita dalla casetta dei fantasmi */
  if(turno<7)
    {
      if(turno<5)
        return SU;

      nodo_a = grafo[13];
      nodo_da = grafo[13];

      colonna_old = -1;
      colonna = nodo_a.colonna;

      riga_old = -1;
      riga = nodo_a.riga;
      return DESTRA;
    }

  /* Sono su un nodo? */
  int inx = indice_nodo_cella(riga,colonna);
  //fprintf(f,"R: %d, C: %d, inx:%d\n",riga,colonna,inx);


  /* cella su cui si trova e cella precedente */
  cella c, pr;
  c.riga = riga;
  c.colonna = colonna;
  pr.riga = riga_old;
  pr.colonna = colonna_old;

  if(inx<0)
    {
      /*_____________________________________________________
       *|  Non sono su un nodo. Il percorso è obbligato
       *| sono definiti: nodo_da, nodo_a, x e y _old
       *| satbilisce il prossimo passo come l'unico
       *| possibile in modo che non sia ne un muro ne la
       *| posizione attuale
       */

      ld = prossima_cella(c, pr, lab);

      colonna_old = colonna;
      riga_old = riga;

      return ld;
    }
  else
    {
      /*_____________________________________________________
       *|
       *| Sono su un nodo, devo decidere il prossimo
       *| edge su cui muovermi
       *| se non ci sono stati errori inx ==  nodo_a.indice
       *| in base alla posizione di tuki.
       *| Al termine di questo blocco devo:
       *| nodo_da <- noda_a
       *| nodo_a <- nodo_dest
       */

      if(inx != nodo_a.indice)
```

```
  {
    //exit(0);
  }

/* Nodo temporaneo */
nodo n_d;

/* calcola la distanza tra i prossimi nodi e tuki */
double d_sx = 10000, d_dx = 10000, d_su = 10000, d_giu = 10000;

cella tuki,clyde;
tuki.riga = p.tuki_y;
tuki.colonna = p.tuki_x;
clyde.riga  = p.clyde_y;
clyde.colonna = p.clyde_x;

double ds = distanza_celle(tuki,clyde);
if(ds <= 8)
  {
    tuki.riga = 28;
    tuki.colonna = 3;
  }

cella s;

//fprintf(f,"1) nodo_a.indice %d\n",nodo_a.indice);

if(nodo_a.n_sx >= 0)
  {
    n_d = grafo[nodo_a.n_sx];
    if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
      {
        s.riga = n_d.riga;
        s.colonna = n_d.colonna;
        d_sx = distanza_celle(s, tuki);
      }
  }
if(nodo_a.n_dx >= 0)
  {
    n_d = grafo[nodo_a.n_dx];
    if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
      {
        s.riga = n_d.riga;
        s.colonna = n_d.colonna;
        d_dx = distanza_celle(s, tuki);
      }
  }
if(nodo_a.n_su >= 0)
  {
    n_d = grafo[nodo_a.n_su];
    if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
      {
        s.riga = n_d.riga;
        s.colonna = n_d.colonna;
        d_su = distanza_celle(s, tuki);
      }
  }
if(nodo_a.n_giu >= 0)
  {
    n_d = grafo[nodo_a.n_giu];
    if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
      {
        s.riga = n_d.riga;
        s.colonna = n_d.colonna;
        d_giu = distanza_celle(s, tuki);
      }
  }
    nodo_da =  nodo_a;

int ind;
/* cerco il nodo più vicino a tuki */
if(d_sx<=d_dx && d_sx<=d_su && d_sx<=d_giu)
  {
    ind = nodo_a.n_sx;
    nodo_a = grafo[ind];

    ld = SINISTRA;
  }else if(d_dx<=d_sx && d_dx<=d_su && d_dx<=d_giu)
  {
    ind = nodo_a.n_dx;
    nodo_a = grafo[ind];

    ld = DESTRA;
```

```
        } else if(d_su<=d_sx && d_su<=d_dx && d_su<=d_giu)
        {
          ind = nodo_a.n_su;
          nodo_a = grafo[ind];

          ld = SU;
        } else if(d_giu<=d_sx && d_giu<=d_dx && d_giu<=d_su)
        {
          ind = nodo_a.n_giu;

          nodo_a = grafo[ind];

          ld = GIU;
        }

      colonna_old = colonna;
      riga_old = riga;

      if(nodo_da.indice == nodo_a.indice)
        {
          //exit(0);
        }

      return ld;
    }
  return ld;
}
```

APPENDICE: FONDAMENTI DI LATINO

I sostantivi

Il latino è una lingua declinata. Questo significa che ogni sostantivo nella frase cambia in base al ruolo che svolge.
Ciò che cambia non è l'intera parola, ma solo la sua parte finale.

Ad esempio, consideriamo il sostantivo "ragazza" in queste due frasi:

1. La ragazza è bella
2. Ho visto la ragazza

Nella prima frase la ragazza è il soggetto.
Nel secondo, la ragazza non è più il soggetto. Sono io il soggetto e lei è l'oggetto su cui ricade la mia azione (complemento oggetto).
Sebbene la parola "ragazza" svolga due ruoli diversi nelle due frasi, la parola non cambia, spetta al lettore comprendere la differenza.

Consideriamo ora la traduzione latina:

1. Puella pulchra est
2. Puellam vidi

Puella significa "ragazza".
Come potete vedere, nelle due frasi, la parte finale della parola è cambiata.
Questo è l'effetto della declinazione: a seconda del ruolo che svolge nella frase, un nome viene declinato in diversi "casi".

I casi

Esistono sei casi ciascuno dei quali rappresenta uno o più ruoli che un sostantivo può assumere:

1. Nominativo: soggetto
   ex. "**La ragazza** è bella" -> *Puella pulchra est*
2. Genitivo: complemento di specificazione
   ex. "Questo è il libro **della ragazza**" -> *Hic puellae liber est*
3. Dativo: complemento di termine
   ex. "Regalo una penna **alla ragazza**" -> *calamum puellae dono*
4. Accusativo: complemento oggetto
   ex. "Ho visto **la ragazza**" -> *Puellam vidi*
5. Vocativo: complemento di vocazione
   ex. "Vieni qui per favore, **ragazza**" -> *Veni, puella*
6. Ablativo: altri complementi
   ex. "Sono andato al mare **con la ragazza**" -> *Ab litori, cum puella ivi*

Le declinazioni

Sembra abbastanza facile, giusto? A seconda del ruolo che un sostantivo svolge nella frase, cambiamo la sua fine e il gioco è fatto.
Sfortunatamente, non è così facile come può sembrare. In realtà non esiste un solo modo per declinare un nome nei sei casi, ma ... cinque modi diversi.
Ognuno di questi modi corrisponde a una declinazione.

In latino ogni nome appartiene a una declinazione.
Paragoniamolo con un linguaggio di programmazione orientato agli oggetti.
Immaginiamo che il *sostantivo* sia una classe astratta con un metodo *declino (caso)* non implementato che restituisce il nome declinato nel caso specificato.

Il latino ha cinque classi che derivano da quella classe astratta e che implementano il metodo "declino" in modo diverso.

Come detto sopra, ci sono cinque declinazioni.
Una parola che appartiene a una specifica declinazione verrà declinata nei sei casi, seguendo le regole di tale declinazione.
Ad esempio, il genitivo della parola "puella", che appartiene alla prima declinazione, è "puellae", dove **ae** è la desinenza che caratterizza il genitivo di ogni parola che appartiene alla prima declinazione.

Prima declinazione

Singolare	Plurale
Cella	Cellae
Cellae	Cellarum
Cellae	Cellis
Cellam	Cellas
Cella	Cellae
Cella	Cellis

Seconda declinazione

Singolare	Plurale
Nodus	Nodi
Nodi	Nodorum
Nodo	Nodis
Nodum	Nodos
Node	Nodi
Nodi	Nodis

Terza declinazione

Singolare	Plurale
Colligatio	Colligationes
Colligationis	Colligationum
Colligationi	Colligationibus
Colligationem	Colligationes
Colligatio	Colligationes
Colligatione	Colligationibus

Quarta declinazione

Singolare	Plurale
Manus	Manus
Manus	Manuum
Manui	Manibus
Manum	Manus
Manus	Manus
Manu	Manibus

Quinta declinazione

Singolare	Plurale
Res	Res
Rei	Rerum
Rei	Rebus
Rem	Res
Res	Res
Re	Rebus

Gli elementi della libreria

La parola *ager* in Latino significa 'campo', nel senso di campo di fiori o campo di grano. Ager appartiene alla terza

declinazione quindi, come avrete imparato, *agri* significa 'del campo'.

```
typedef enum {SX,DEORSUM,DX,SURSUM,FIXUS} versus
```

- **versus** decl. 4: direzione
- **sx** sta per sinistra decl. 1: sinistra
- **deorsum** avverbio: giu
- **dx** sta per dextra decl. 1: destra
- **sursum** avv: su
- **fixo** aggettivo: fermo

```
enum genus {MURO,ALTRO}
```

- **genus** decl. 3: tipo
- **muro** (Italiano)
- **altro** (Italiano)

```
typedef enum genus rei_genus
```

- **rei_genus**: tipo dell'oggetto

Grafo come una lista di archi:

```
typedef struct { int linea; int columna; int index;
int ianua[PORTE]; } agri_Vertex;
```

- **vertex** decl. 3: vertice
- **linea** decl. 1: riga
- **columna** decl. 1: colonna
- **index** decl. 3: indice
- **ianua** decl. 1: porta

```
typedef struct colligatio { agri_Vertex ab, ad;
versus discessus, meta; int longitudo; }
agri_Colligatio;
```

- **colligatio** decl. 3: arco

- **discessus** decl. 4: partenza
- **meta** decl. 1: destinazione
- **longitudo** decl. 3: lunghezza

```
typedef struct membrum { agri_Colligatio colligatio; struct membrum * next; } agri_Membrum;
```

- **colligatio** decl. 3: arco
- **membrum** decl. 2: elemento

```
typedef agri_Membrum * agri_Colligationes_Colligatae;
```

- **colligationes colligatae**: archi connessi

```
void agri_Colligationem_insero
```

- **colligationem insero**: inserisco arco

```
int agri_Verticem_quaero
```

- **verticem quaero**: cerco un vertice

Grafo come lista di vertici:

```
typedef agri_Vertex * agri_Verticum_Dispositio;
```

- **verticum dispositio**: array di vertici

```
agri_Vertex agri_Verticem_creo(int index, int linea, int columna);
```

- **verticem creo**: creo un vertice

```
typedef struct nodus_coda { int index; double prio; struct nodus_coda * post; } Nodus;
```

- **nodus** decl.2: nodo (elemento della coda)
- **index** decl. 3: indice

- **prio** sta per priorsum avv: priorità
- **coda** decl.1: coda

```
typedef Nodus * Ordo;
```

- **ordo** decl.3: lista

```
typedef int * agri_Via;
```

- **via** decl.1: via, cammino

```
void Ordo_amoveo_nodus(Ordo * l, int index);
```

- **amoveo nodus**: rimuovo nodo

```
void Ordo_insero_nodus(Ordo * l,int index, double prio);
```

- **insero nodus**: inserisco nodo

```
int Ordo_pop(Ordo * pOrdo);
```

- (Inglese) pop dell'elemento in testa alla lista

```
agri_Via agri_astar(int start, int goal, agri_Vertex * agri_Vertices_Colligati, double (*spatium)(int ab, int ad), double (*euristica)(int ab, int ad));
```

- (Inglese) algoritmo Astar

```
int index_nodus_cella(int riga, int col, agri_Vertex * av);
```

- **index nudus cella**: indice nodo cella

Grafo come una lista di celle:

```
typedef struct dato { int visitata; rei_genus rei;
```

```
} Attributi;
```

- **Attributi** part. perf.: attributi
- **visitata** agg: visitata

```
typedef struct cella { Attributi d; struct cella* dextra; struct cella* deorsum; struct cella* sinistra; struct cella* sursum; } agri_Cella;
```

- **Cella** decl.1: cella

```
typedef agri_Cella* agri_Tabella;
```

- **tabella** decl.1: tabella, griglia

```
typedef struct nodo { agri_Cella * locus; struct nodo * next; struct nodo * prev; }agri_Passo;
```

- **passo** da passus decl.2: passo
- **locus** decl.2: luogo, posizione

```
typedef agri_Passo* agri_Iter;
```

- **iter** decl.3: percorso

```
void agri_creo_Tabellam(agri_Tabella* g);
```

- **creo tabellam**: creo tabella

```
agri_Tabella        agri_addo_Tabellam(agri_Tabella tabella,Attributi d,versus dir);
```

- **addo tabellam**: aggiungo tabella

```
void    agri_colligo_Cellas(agri_Cella    *    da, agri_Cella * a, versus da_a);
```

- **colligo cellas**: connetto celle

```
void agri_creo_Iter(agri_Iter* p_camm);
```

- **creo iter**: creo percorso

```
void agri_addo_Iter(agri_Iter * p_camm, agri_Cella
* p_cella);
```

- **addo iter**: aggiungo percorso

```
agri_Cella*  agri_rivela_Cella(agri_Iter  ap,versus
ricerca);
```

- **rivela cella**: rivela cella

```
int   agri_muto(agri_Colligationes_Colligatae   ge,
agri_Verticum_Dispositio *a);
```

- **muto**: trasformo

```
int agri_dispono(agri_Colligationes_Colligatae list
,agri_Colligationes_Colligatae * array);
```

- **dispono**: dispongo

```
int Verticem_quaero(agri_Verticum_Dispositio v, int
index, int amplitudo);
```

- **verticem quaero**: cerco un vertice

```
versus agri_Versum_inverto(versus v);
```

- **versum inverto**: inverto direzione

```
int compar(const void * a, const void * b);
```

- **compar** sta per comparo: confronto

APPENDICE: CODICE COMPLETO LIBRERIA AGRI

Listato libagri.h

```
/*_____________________________________________
* FILE: libagri.h
*/

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

#define NNODI 44
#define PORTE 4
#define INFINITO 9999

typedef enum  {SX,DEORSUM,DX,SURSUM,FIXO} versus;
enum genus {MURO,ALTRO};
typedef enum genus rei_genus;

/*** TIPI COMUNUI ***/

typedef struct {
  int linea;
  int columna;
  int index;
  int ianua[PORTE];
} agri_Vertex;

/*** GRAFO COME LISTA DI ARCHI ***/

/*
   ITA: Arco orientato e pesato
   ENG: diricted and weighted edge ("colligatio" means edge)
*/
typedef struct colligatio {
  /*
     ITA: vertici collegati dall'arco
     ENG: vertices connected by the edge
  */
  agri_Vertex ab, ad;
   /*
    ITA: attributi
    ENG: attributes
   */
  versus discessus, meta;
  int longitudo;
} agri_Colligatio;

/*
   ITA: Elemento della lista di archi
   ENG: Edges list item
*/
typedef struct membrum {
  agri_Colligatio colligatio;
  struct  membrum *  next;
} agri_Membrum;

typedef agri_Membrum * agri_Colligationes_Colligatae;

/*
  ITA: Inserisce l'arco in testa alla lista
  ENG: inserts an edge on top of the list
*/
void agri_Colligationem_insero
(agri_Colligationes_Colligatae * pg, agri_Colligatio colligatio);

/* ITA: Cerca tra gli elementi del grafo se uno degli archi è connesso
   ad un vertice in linea e columna. Se lo trova torna l'indice del
   vertice, altrimenti -1
   ENG: Looks into the graph for a vertex whose row and column attributes
   are equal to linea and columna
```

```
*/
int agri_Verticem_quaero
(agri_Colligationes_Colligatae g, int linea, int columna);

void agri_libero(agri_Colligationes_Colligatae g);

/*** GRAFO COME ARRAY DI VERTICI ***/

typedef agri_Vertex * agri_Verticum_Dispositio;

agri_Vertex agri_Verticem_creo(int index, int linea, int columna);

typedef struct nodus_coda
{
  //Riferimento al nodus del grafo
  int index;

  //Priorità
  double prio;

  //Navigazione coda
  struct nodus_coda * post;

} Nodus;

//Tipo per la coda di priorità
typedef Nodus * Ordo;
//Tipo per il cammino finale
typedef int * agri_Via;

/*_____________________________________________
 *
 * Inserimento ordinato in base alla priorità
 */
void Ordo_amoveo_nodus(Ordo * l, int index);

void Ordo_insero_nodus(Ordo * l,int index, double prio);

int Ordo_pop(Ordo * pOrdo);

agri_Via agri_astar(int start, int goal,
                    agri_Vertex * agri_Vertices_Colligati,
                    double (*spatium)(int ab, int ad),
                    double (*euristica)(int ab, int ad),
                    int nmembri
                    );

int agri_breadthfirstsearch(int start,
                    agri_Vertex * agri_Vertices_Colligati,
                    int (*visitatus)(int ),
                    int nmembri
                    );

/*_____________________________________________________________
 *|
 *| Torna l'index del nodus presente in (riga, colonna)
 *| Se non è presente alcun nodus (il fantasma è su un ramo)
 *| torna -1
 */
int index_nodus_cella
(int riga, int col, agri_Vertex * av,int nmembri);

/*** GRAFO COME LISTA DI CELLE  ***/

/*
  ITA: Attributi della cella
  ENG: Cell attributes
 */
typedef struct dato
{
  int visitata;
  rei_genus rei;
} Attributi;

/*
  ITA: vicini di ogni cella
  ENG: cell's neighbords
 */
typedef struct cella {
  Attributi d;
  struct cella* dextra;
  struct cella* deorsum; //GIU
  struct cella* sinistra;
```

```
  struct cella* sursum; //SU
} agri_Cella;

/*
  ITA: Tabellam composto da celle
  ENG: Graph composed by cells
 */
typedef agri_Cella* agri_Tabella;

/*
  ITA: elementi della lista dei passi
  ENG: elements of the step list
 */
typedef struct nodo
{
  agri_Cella * locus;
  struct nodo * next;
  struct nodo * prev;
}agri_Passo;

/*
  ITA: lista dei passi
  ENG: step list
 */
typedef agri_Passo* agri_Iter;


/*
  ITA: inizializza un grafo vuoto
  ENG: initialize an empty graph
 */
void agri_creo_Tabellam(agri_Tabella* g);
/*
  ITA: Aggiunge al grafo una nuova cella con attributi d,
  collegandola alla direzione
  dir cella corrente
  ENG: Adds a new cell, with the attributes d, to the graph,
  linking it to the
  direction dir of the current cell
 */
agri_Tabella agri_addo_Tabellam
(agri_Tabella tabella,Attributi d,versus dir);

/*
  ITA: collega le cella da e a lungo la direzione da ---> a
 */
void agri_colligo_Cellas(agri_Cella * da, agri_Cella * a, versus da_a);

/*
  ITA: inizializza una lista di passo puntata da p_camm cammimo vuoto
  ENG: initialize an empty list of steps
 */
void agri_creo_Iter(agri_Iter* p_camm);
/*
  ITA: aggiunge una Cella nella lista puntata da p_camm dei passi compiuti
  ENG: adds a cell in the list pointed to by p_camm of the steps performed
 */
void agri_addo_Iter(agri_Iter * p_camm, agri_Cella * p_cella);
/*
  ITA:
 */
agri_Cella* agri_rivela_Cella(agri_Iter ap,versus ricerca);

/*** TRAFFORMAZIONI ***/

/*
  ITA: da un grafo di archi ad un grafo di vertici
 */
int  agri_muto
(agri_Colligationes_Colligatae ge, agri_Verticum_Dispositio *a);

int agri_dispono
(agri_Colligationes_Colligatae list ,agri_Colligationes_Colligatae * array);

int Verticem_quaero(agri_Verticum_Dispositio v, int index, int amplitudo);

versus agri_Versum_inverto(versus v);

int compar(const void * a, const void * b);
```

Listato libagri.c

```
/*_____________________________________________________________
* FILE: libagri.c
*/

#include "libagri.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/*** GRAFO COME ARRAY DI VERTICI ***/

agri_Vertex agri_Verticem_creo(int index, int linea, int columna)
{
  agri_Vertex v;
  v.index =  index;
  v.linea = linea;
  v.columna = columna;
  for(int i = 0;i<PORTE; i++)
    v.ianua[i] = -1;
  return v;
}

void agri_Colligationem_insero
(agri_Colligationes_Colligatae * pg, agri_Colligatio colligatio)
{

  agri_Colligationes_Colligatae cg = *pg;

  if(colligatio.ab.index == colligatio.ad.index)
    return;

   while(cg)
    {
      /*
         ITA: l'arco è già nel grafo
         ENG: the edge is already in the graph
       */
      if(cg->colligatio.ab.index == colligatio.ab.index
         && cg->colligatio.ad.index == colligatio.ad.index)
        return;
      cg = cg->next;
    }

  agri_Membrum * aux = malloc(sizeof(agri_Membrum));
  if(aux == 0) exit(1);
  aux -> colligatio = colligatio;
  aux -> next = *pg;
  *pg = aux;
}

int agri_Verticem_quaero
(agri_Colligationes_Colligatae g, int linea, int columna)
{
  while(g)
    {
      if(g->colligatio.ab.linea == linea &&
           g->colligatio.ab.columna == columna)
        return g->colligatio.ab.index;
      if(g->colligatio.ad.linea == linea &&
           g->colligatio.ad.columna == columna)
        return g->colligatio.ad.index;
      g = g->next;
    }
  return -1;
}

void agri_libero(agri_Colligationes_Colligatae g)
{
  agri_Colligationes_Colligatae g_t;
  while(g)
    {
      g_t = g;
      g = g->next;
      free(g);
    }
}

void Ordo_amoveo_nodus(Ordo * l, int index)
{

  while (*l) {
    if ((*l)->index == index)
```

```
      break;
    l = &(*l)->post;
  }
  if (*l) {
    Nodus* aux = *l;
    *l = (*l)->post;
    free(aux);
  }
}

void Ordo_insero_nodus(Ordo * l,int index, double prio)
{

  Nodus * aux = (Nodus *)malloc(sizeof(Nodus));
  aux->index =  index;
  aux->prio = prio;

  /* il nodus è già in Ordo */
  Ordo_amoveo_nodus(l,  index);

  while (*l) {
    if ((*l)->prio < prio)
      break;
    l = &(*l)->post;
  }

  aux->post = *l;
  *l = aux;
}

int Ordo_pop(Ordo * pOrdo)
{
  /* puntatore al prio nodus */
  Ordo candidato = *pOrdo;

  int n = candidato->index;

  /* aggiorna la Ordo */
  (*pOrdo) = candidato->post;

  free(candidato);

  return n;
}

agri_Via agri_astar
(int start, int goal,
 agri_Vertex * agri_Vertices_Colligati,
 double (*spatium)(int ab, int ad),
 double (*euristica)(int ab, int ad),
 int nmembri
 )
{
  if(start==goal)
    {
      return 0;
    }
  double fscore[nmembri];
  double gscore[nmembri];
  int precedente[nmembri];

  //Nodi in valutazione per il path da start a goal
  Ordo candidati = 0;

  for(int i=0;i<nmembri;i++)
    {
      fscore[i] = INFINITO;
      gscore[i] = INFINITO;
    }
  gscore[start] = 0;
  fscore[start]=euristica(start,goal);

  Ordo_insero_nodus(&candidati,start,(1./fscore[start]));

  while(candidati != 0)
    {
      fflush(stdout);
      int corrente = Ordo_pop(&candidati);

      /* Arrivato al nodus goal torna il cammino */
      if(corrente == goal)
        {
          fflush(stdout);
```

```
          int i = 0;
          int aux[nmembri];
          do{
            aux[i]=corrente;
            corrente=precedente[corrente];
            i++;
          }while(corrente != start);

          agri_Via percorso = malloc((i+1)*sizeof(int));

          int j =0 ;
          for(int k = i-1;k >=0 ;k--,j++)
            {
              percorso[j] = aux[k];

            }
          // Segnale di fine percorso
          percorso[j]=-1;
          return percorso;
        }

      /* Nel labirinto di Pac-Man ci sono al massimo 4 vicini */
      int vicino[PORTE];
      for(int i=0; i<PORTE; i++)
        {
          vicino[i] = agri_Vertices_Colligati[corrente].ianua[i];
        }

      for(int i =0; i<PORTE; i++)
        {
          int iv = vicino[i];

          /* ogni nodus è predisposto per 4 vicini, ma molti nodi ne hanno
             solo 3*/
          if(iv == -1)continue;

          double d =  spatium(corrente, iv);
          double tent_gscore = gscore[corrente]+d;

          if(tent_gscore <= gscore[iv])
            {
              /* Trovato cammino migliore a questo nodus passando per corrente*/
              precedente[iv] = corrente;
              gscore[iv] = tent_gscore;
              fscore[iv] = gscore[iv] +  euristica(iv, goal);

              Ordo_insero_nodus(&candidati,iv, ( 1./fscore[iv]));

              fflush(stdout);
            }
        }
    }
  return 0;
}

int agri_breadthfirstsearch(int start,
                    agri_Vertex * agri_Vertices_Colligati,
                    int (*visitatus)(int ),
                    int nmembri
                    )
{
  static int iter;

  int precedente[nmembri];

  //Nodi in valutazione per il path da start a goal
  Ordo candidati = 0;


  Ordo_insero_nodus(&candidati,start,1);

  while(candidati != 0)
    {
      int corrente = Ordo_pop(&candidati);

      /* Arrivato al nodus goal torna il cammino */
      if(!visitatus(corrente) && corrente!=start)
        {
          return corrente;
        }

      int vicino[PORTE];
      for(int i=0; i<PORTE; i++)
```

```
        {
          vicino[i] = agri_Vertices_Colligati[corrente].ianua[i];
        }

      for(int i =0; i<PORTE; i++)
        {
          int iv = vicino[i];

          if(iv == -1)continue;

          precedente[iv] = corrente;

          Ordo_insero_nodus(&candidati,iv,1);
        }
    }


  return 0;
  }

int index_nodus_cella
(int riga, int col, agri_Vertex * agri_Vertices_Colligati,int nmembri)
{
  for(int i=0; i<nmembri;i++)
    {
      if(agri_Vertices_Colligati[i].columna == col
         && agri_Vertices_Colligati[i].linea == riga)
        return agri_Vertices_Colligati[i].index;
    }
  return -1;
}

/*** GRAFO COME LISTA DI CELLE  ***/

void agri_creo_Tabellam(agri_Tabella* g)
{
  *g = NULL;
}

/**
 *Add a cell to the agri_Tabella after checking that a cell
 *was already present at the same poistion.
 *For each cell of the actual Tuki agri_Iter is ssursumposed
 *to exist another 4 cells at the ajacent direzione
 *This functions.
 *The existence of a possible unlinked cell in the 'd'
 *direzione is tested scann2ing the agri_Iter_list untill a
 *cell
 */
agri_Tabella agri_addo_Tabellam
(agri_Tabella tabella,Attributi d,versus dir)
{
  agri_Cella* n=(agri_Cella*)malloc(sizeof(agri_Cella));
  n->d=d;

  switch(dir)
    {
    case DEORSUM:
      tabella->sursum=n;
      n->deorsum=tabella;
      break;
    case SURSUM:
      tabella->deorsum=n;
      n->sursum=tabella;
      break;
    case DX:
      tabella->dextra=n;
      n->sinistra=tabella;
      break;
    case SX:
      tabella->sinistra=n;
      n->dextra=tabella;
      break;
    }
  return n;
}

void agri_colligo_Cellas
(agri_Cella * da, agri_Cella * a, versus da_a)
{
  switch(da_a)
    {
    case DEORSUM:
```

```
      da->sursum=a;
      a->deorsum=da;
      break;
    case SURSUM:
      da->deorsum=a;
      a->sursum=da;
      break;
    case DX:
      da->dextra=a;
      a->sinistra=da;
      break;
    case SX:
      da->sinistra=a;
      a->dextra=da;
      break;
    }
}

void agri_creo_Iter(agri_Iter* p){
  *p=0;
}
void agri_addo_Iter(agri_Iter* l, agri_Tabella g)
{
  agri_Passo* aux = (agri_Passo*)malloc(sizeof(agri_Passo));
  aux->prev = *l;
  aux->next = NULL;
  aux->locus = g;
  if(*l)
    (*l)->next=aux;
  *l = aux;
}

/*
  Cerca una eventuale cella nella direzione dir che sia già stata esplorata
  lungo il cammino ma che non sia ancora stata connessa alla cella corrente
  del cammino
 */
agri_Cella* agri_rivela_Cella(agri_Iter ap,versus dir){
  int R=0,U=0;
  int found=0;

  while(ap->prev){
    if(ap->locus->dextra ==ap->prev->locus) R++;
    if(ap->locus->sinistra == ap->prev->locus) R--;
    if(ap->locus->sursum == ap->prev->locus) U++;
    if(ap->locus->deorsum == ap->prev->locus) U--;
    /*check neighbor condition*/
    //DX
    if(dir==DX && R==1 && U==0)
      {
        return ap->prev->locus;
      }
    //R-U
    if(dir==DX && (R==1) && U==+1)
      {
        return ap->prev->locus->deorsum;
      }
    //R-D
    if(dir==DX&&(R==1)&&U==-1)
      {
        return ap->prev->locus->sursum;
      }

    //Sinistra
    //L
    if(dir==SX&&R==-1&&(U==0))
      {
        return ap->prev->locus;
      }
    //L-U
    if(dir==SX&&(R==-1)&&U==+1)
      {
        return ap->prev->locus->deorsum;
      }
    //L-D
    if(dir==SX&&(R==-1)&&U==-1)
      {
        return ap->prev->locus->sursum;
      }

    //Sursum
    //SU
    if(dir==DEORSUM&&R==0&&U==1)
```

```
      {
        return ap->prev->locus;
      }
    //U-L
    if(dir==DEORSUM&&(R==-1)&&U==+1)
      {
        return ap->prev->locus->dextra;
      }
    //U-R
    if(dir==DEORSUM&&(R==1)&&U==+1)
      {
        return ap->prev->locus->sinistra;
      }

    //Deorsum
    // D
    if(dir==SURSUM&&R==0&&U==-1)
      {
        return ap->prev->locus;
      }
    //D-L
    if(dir==SURSUM&&(R==-1)&&U==-1)
      {
        return ap->prev->locus->dextra;
      }
    //D-R
    if(dir==SURSUM&&(R==1)&&U==-1)
      {
        return ap->prev->locus->sinistra;
      }

    ap=ap->prev;
  }
  return NULL;
}

/*
  ITA: da un grafo di archi ad un grafo di vertici
 */
int  agri_muto
(agri_Colligationes_Colligatae g,agri_Verticum_Dispositio* d)
{
  //Trasformo prima la losta di archi in un array di archi
  agri_Colligationes_Colligatae array;
  int n = agri_dispono(g , &array);

  //Array di vertici
  int sz = 0;
  agri_Verticum_Dispositio v = malloc(2*n*sizeof(agri_Vertex));

  for(int i=0; i<n; i++)
    {
      int ix_ab = (array+i)->colligatio.ab.index;
      int ix_ad = (array+i)->colligatio.ad.index;
      int ix = Verticem_quaero(v,ix_ab,sz);

      if(ix<0)
        {
          memcpy(v+sz,&((array+i)->colligatio.ab),sizeof(agri_Vertex));
          ix = sz;
          sz++;
        }
      int iy = Verticem_quaero(v,ix_ad,sz);

      if(iy<0)
        {
          memcpy(v+sz,&((array+i)->colligatio.ad),sizeof(agri_Vertex));
          iy = sz;
          sz++;
        }

      /* collego i vertici */
      if ((array+i)->colligatio.discessus != -1)
        {
          //Assegna la porta in base alla direzione
          (v+ix)->ianua[(array+i)->colligatio.discessus]=ix_ad;
          (v+iy)->ianua[agri_Versum_inverto((array+i)->colligatio.meta)]=ix_ab;
        }
      else
        {
          //Cerco la prima porta libera
          int k = 0;
          while((v+ix)->ianua[k]!=-1)k++;
```

```
          (v+ix)->ianua[k] = ix_ad;

          k = 0;
          while((v+iy)->ianua[k]!=-1)k++;
          (v+iy)->ianua[k] = ix_ab;
        }
    }
  qsort(v, sz, sizeof(agri_Vertex),&compar);
  *d = v;
  return sz;
}

int compar(const void * a, const void * b)
{
  return (((agri_Vertex*)a)->index-((agri_Vertex*)b)->index);
}

int Verticem_quaero(agri_Verticum_Dispositio v, int index, int size)
{
  for(int i=0; i<size;i++)
    {
      if((v+i)->index==index)
        return i;
    }

  return -1;
}

/*
  ITA: crea un'array di vertici partendo da una lista collegata di vertici
 */
int agri_dispono
(agri_Colligationes_Colligatae list ,agri_Colligationes_Colligatae * array)
{
  agri_Colligationes_Colligatae g = list;

  int n = 0;
  while(g)
    {
      n++;
      g = g->next;
    }
  if (n==0) return 0;

  (*array) = malloc(n*sizeof(agri_Membrum));
  g = list;

  for(int i=0; i<n;i++)
    {
      memcpy(*array+i,g,sizeof(agri_Membrum));

      if(i<n+1)
        (*array+i)->next = (*array+i+1);

      g = g->next;
    }
  return n;
}

versus agri_Versum_inverto(versus v)
{
  if (v == SX) return DX;
  if (v == DX) return SX;
  if (v == DEORSUM) return SURSUM;
  if (v == SURSUM) return DEORSUM;
  if (v == FIXO) return FIXO;
  return FIXO;
}
```

APPENDICE: CODICE COMPLETO AGENTS DI PAC-MAN

Listato gioca_tuki_vuoto.c

```
/*_____________________________________________
* FILE: gioca_tuki_vuoto.c
*/

#include "tuki5_modello.h"
#include <stdio.h>
#include <unistd.h>

direzione gioca_tuki(posizioni p, oggetto **labx)
{
  static direzione ld=SINISTRA;

  return ld;
}
```

Listato gioca_tuki_random.c

```
/*_____________________________________________________________
 *
 * FILE: gioca_tuki_random.c
 */
#include "tuki5_modello.h"
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

bool oggetto_accessibile(oggetto s)
{
  if(s == 'J' || s == 'U' || s == 'V')
    return true;
  else
    return false;
}

direzione gioca_tuki(posizioni posi, oggetto **labx)
{
  static direzione ld = SINISTRA;
  static bool init = false;
  if(!init)
    {
      srand(time(0));
      init = true;
    }

  int i = posi.tuki_y;
  int j = posi.tuki_x;

  oggetto s,d,a,b;
  s = labx[i][j-1]; //sinistra, left
  d = labx[i][j+1]; //destra, right
  a = labx[i-1][j]; //su, up
  b = labx[i+1][j]; //giu, down

  int x = posi.tuki_x;
  int y = posi.tuki_y;
  int x_g[4];
  int y_g[4];
  x_g[0] = posi.blinky_x;
  x_g[1] = posi.pinky_x;
  x_g[2] = posi.inky_x;
  x_g[3] = posi.clyde_x;

  y_g[0] = posi.blinky_y;
```

```
y_g[1] = posi.pinky_y;
y_g[2] = posi.inky_y;
y_g[3] = posi.clyde_y;

char s_g = 0, d_g = 0,a_g = 0,b_g = 0;
for (int ig = 0; ig<4; ig++)
  {
    s_g = s_g || ( x_g[ig] < x) && ( x - x_g[ig] <=2 )  && (y == y_g[ig]);
    s_g = s_g || ((x_g[ig] == x-1) &&  (y_g[ig] == y+1));
    s_g = s_g || ((x_g[ig] == x-1) &&  (y_g[ig] == y-1));
    d_g = d_g || ( x_g[ig] > x) && (x_g[ig] - x <= 2) && (y == y_g[ig]);
    d_g = d_g || ((x_g[ig] == x+1) &&  (y_g[ig] == y+1));
    d_g = d_g || ((x_g[ig] == x+1) &&  (y_g[ig] == y-1));
    a_g = a_g || (x == x_g[ig]) && ( y >  y_g[ig]) && ( y - y_g[ig] <=2);
    a_g = a_g || ((y == y_g[ig] + 1) && (x_g[ig] == x+1));
    a_g = a_g || ((y == y_g[ig] + 1) && (x_g[ig] == x-1));
    b_g = b_g || (x == x_g[ig]) && ( y <  y_g[ig]) && ( y_g[ig]- y <=2);
    b_g = b_g || ((y == y_g[ig] - 1) && (x_g[ig] == x+1));
    b_g = b_g || ((y == y_g[ig] - 1) && (x_g[ig] == x-1));
  }

if((s_g || d_g || a_g || b_g) && FUGA)
  {
    direzione esc[4];
    for(int i=0;i<4;i++) esc[i]=FERMO;
    int ki = 0; //direzioni buone, good directions

    if(oggetto_accessibile(s) && !s_g)
      {
        esc[ki] = SINISTRA;
        ki++;
      }
    if(oggetto_accessibile(a) && !a_g)
      {
        esc[ki] = SU;
        ki++;
      }
    if(oggetto_accessibile(d) && !d_g)
      {
        esc[ki] = DESTRA;
        ki++;
      }
    if(oggetto_accessibile(b) && !b_g)
      {
        esc[ki] = GIU;
        ki++;
      }
    if(ki == 0)
      {
        return FERMO;
      }
    ld = esc[rand()%ki];

    return ld;
  }

bool trovata_direzione = false;

bool aleatorio = false;

while(!trovata_direzione)
  {
    if(!oggetto_accessibile(s) && ld == SINISTRA)
      {
        ld = rand()%2;
        if(ld == 0)
          ld = SU;
        else
          ld = GIU;
        aleatorio = true;
      }
    else if(!oggetto_accessibile(d) && ld == DESTRA)
      {
        ld = rand()%2;
        if(ld == 0) ld = SU;
        else
          ld = GIU;
        aleatorio = true;
      }
    else if(!oggetto_accessibile(a) && ld == SU)
      {
        ld = rand()%2;
        if(ld == 0) ld = SINISTRA;
```

```
          else
            ld = DESTRA;
          aleatorio = true;
        }
      else if(!oggetto_accessibile(b) && ld == GIU)
        {
          ld = rand()%2;
          if(ld == 0) ld = SINISTRA;
          else
            ld = DESTRA;
          aleatorio = true;
        }
      else trovata_direzione = true;
    }

  if(aleatorio) return ld;

  if(oggetto_accessibile(a) && ld !=SU && ld!=GIU)
    {
      int sv = rand()%10;
      if(sv>=5)
        ld = SU;
    }
  if(oggetto_accessibile(b) && ld !=GIU && ld!=SU)
    {
      int sv = rand()%10;
      if(sv>=5)
        ld = GIU;
    }
  if(oggetto_accessibile(s) && ld !=SINISTRA && ld!=DESTRA)
    {
      int sv = rand()%10;
      if(sv>=5)
        ld = SINISTRA;
    }
  if(oggetto_accessibile(d) && ld !=DESTRA && ld!=SINISTRA)
    {
      int sv = rand()%10;
      if(sv>=5)
        ld = DESTRA;
    }

  return ld;
}
```

Listato gioca_tuki_boustrophedon.c

```
/*_____________________________________________
* FILE: gioca_tuki_boustrophedon.c
*/

#include "tuki5_modello.h"
#include "libagri.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

rei_genus rivela_rei_genus(oggetto contenuto);

agri_Tabella moveo_Cellam(agri_Tabella * ptab, versus dir);

direzione gioca_tuki(posizioni posi, oggetto **labx){

  labx[16][2]='A';
  labx[16][23]='A';
  labx[14][13]='A';
  labx[14][14]='A';

  static int init=0;
  static agri_Tabella g=NULL;
  static agri_Iter p=NULL;
  static agri_Iter l=NULL;

  int i = posi.tuki_y;
  int j = posi.tuki_x;
  oggetto s_ = labx[i][j-1];
  oggetto d_ = labx[i][j+1];
  oggetto a_ = labx[i-1][j];
```

```
oggetto b_ = labx[i+1][j];

if(!init)
  {
    init=1;

    agri_creo_Tabellam(&g);
    agri_Tabella fn=(agri_Tabella)malloc(sizeof(agri_Cella));
    g=fn;
    g->d.visitata=1;
    g->d.rei=ALTRO;

    agri_creo_Iter(&p);
    agri_addo_Iter(&p,g);
  }

int x = posi.tuki_x;
int y = posi.tuki_y;
int x_g[4];
int y_g[4];
x_g[0] = posi.blinky_x;
x_g[1] = posi.pinky_x;
x_g[2] = posi.inky_x;
x_g[3] = posi.clyde_x;

y_g[0] = posi.blinky_y;
y_g[1] = posi.pinky_y;
y_g[2] = posi.inky_y;
y_g[3] = posi.clyde_y;

char s_g = 0, d_g = 0,a_g = 0,b_g = 0;
for (int ig = 0; ig<4; ig++)
  {
    s_g = s_g || ( x_g[ig] < x) && ( x - x_g[ig] <=2 )  && (y == y_g[ig]);
    s_g = s_g || ((x_g[ig] == x-1) &&  (y_g[ig] == y+1));
    s_g = s_g || ((x_g[ig] == x-1) &&  (y_g[ig] == y-1));
    d_g = d_g || ( x_g[ig] > x) && (x_g[ig] - x <= 2) && (y == y_g[ig]);
    d_g = d_g || ((x_g[ig] == x+1) &&  (y_g[ig] == y+1));
    d_g = d_g || ((x_g[ig] == x+1) &&  (y_g[ig] == y-1));
    a_g = a_g || (x == x_g[ig]) && ( y >  y_g[ig]) && ( y - y_g[ig] <=2);
    a_g = a_g || ((y == y_g[ig] + 1) && (x_g[ig] == x+1));
    a_g = a_g || ((y == y_g[ig] + 1) && (x_g[ig] == x-1));
    b_g = b_g || (x == x_g[ig]) && ( y <  y_g[ig]) && ( y_g[ig]- y <=2);
    b_g = b_g || ((y == y_g[ig] - 1) && (x_g[ig] == x+1));
    b_g = b_g || ((y == y_g[ig] - 1) && (x_g[ig] == x-1));
  }

unsigned char c[4];
oggetto cx;

g=p->locus;
if(!g) exit(-1);

cx = a_;

if(g->sursum==NULL)
  {
    agri_Tabella gc=agri_rivela_Cella(p,SU);
    if(gc)
      {
        agri_colligo_Cellas(g,gc,SU);
      }
    else
      {
        Attributi d={0,rivela_rei_genus(cx)};
        agri_Tabella tg=agri_addo_Tabellam(g,d,SU);
      }
  }
else
  {
    g->sursum->d.rei=rivela_rei_genus(cx);
  }

//Deorsum
cx = b_;
if(g->deorsum==NULL)
  {
    agri_Tabella gc=agri_rivela_Cella(p,GIU);
    if(gc)
      {
        agri_colligo_Cellas(g,gc,GIU);
      }
    else
```

```
      {
        Attributi d={0,rivela_rei_genus(cx)};
        agri_Tabella tg=agri_addo_Tabellam(g,d,GIU);
      }
  }
else
  {
    g->deorsum->d.rei=rivela_rei_genus(cx);
  }

//Dextra
cx = d_;
if(g->dextra==NULL)
  {
    agri_Tabella gc=agri_rivela_Cella(p,DESTRA);
    if(gc)
      {
        agri_colligo_Cellas(g,gc,DESTRA);
      }
    else
      {
        Attributi d={0,rivela_rei_genus(cx)};
        agri_Tabella tg=agri_addo_Tabellam(g,d,DESTRA);
      }
  }
else
  {
    g->dextra->d.rei=rivela_rei_genus(cx);
  }

//Sinistra
cx = s_;
if(g->sinistra==NULL)
  {
    agri_Tabella gc=agri_rivela_Cella(p,SINISTRA);
    if(gc)
      {

        agri_colligo_Cellas(g,gc,SINISTRA);
      }
    else
      {
        Attributi d={0,rivela_rei_genus(cx)};
        agri_Tabella tg=agri_addo_Tabellam(g,d,SINISTRA);
      }
  }
else
  {
    g->sinistra->d.rei=rivela_rei_genus(cx);
  }

if((s_g || d_g || a_g || b_g) && FUGA)
  {

    if(g->sinistra->d.rei!=MURO && !s_g)
      {
        moveo_Cellam(&g, SINISTRA);
        g->d.visitata=1;
        agri_addo_Iter(&p,g);
        l=p;
        return SINISTRA;
      }

    if(g->sursum->d.rei!=MURO && !a_g)
      {
        moveo_Cellam(&g, SU);
        g->d.visitata=1;
        agri_addo_Iter(&p,g);
        l=p;
        return SU;
      }

    if(g->dextra->d.rei!=MURO && !d_g)
      {
        moveo_Cellam(&g, DESTRA);
        g->d.visitata=1;
        agri_addo_Iter(&p,g);
        l=p;
        return DESTRA;
      }

    if(g->deorsum->d.rei!=MURO && !b_g)
      {
```

```
          moveo_Cellam(&g, GIU);
          g->d.visitata=1;
          agri_addo_Iter(&p,g);
          l=p;
          return GIU;
        }
    }

  if(g->sinistra->d.visitata==0&&g->sinistra->d.rei!=MURO)
    {
      g=g->sinistra;
      g->d.visitata=1;
      agri_addo_Iter(&p,g);
      l=p;
      return SINISTRA;
    }

  if(g->sursum->d.visitata==0&&g->sursum->d.rei!=MURO)
    {
      g=g->sursum;
      g->d.visitata=1;
      agri_addo_Iter(&p,g);
      l=p;
      return SU;
    }

  if(g->dextra->d.visitata==0&&g->dextra->d.rei!=MURO)
    {
      g=g->dextra;
      g->d.visitata=1;
      agri_addo_Iter(&p,g);
      l=p;
      return DESTRA;
    }

  if(g->deorsum->d.visitata==0&&g->deorsum->d.rei!=MURO)
    {
      g=g->deorsum;
      g->d.visitata=1;
      agri_addo_Iter(&p,g);
      l=p;
      return GIU;
    }

  l=l->prev;
  if(l==NULL) exit(-1);

  agri_addo_Iter(&p,l->locus);

  direzione nd;

  if(g->dextra==l->locus) nd=DESTRA;
  else if(g->sinistra==l->locus) nd=SINISTRA;
  else if(g->sursum==l->locus) nd=SU;
  else if(g->deorsum==l->locus) nd=GIU;

  return nd;
}

rei_genus rivela_rei_genus(oggetto code)
{
  if(code != 'J' && code != 'U' && code != 'V')
    {
      return MURO;
    }

  return ALTRO;
}

agri_Tabella moveo_Cellam(agri_Tabella * ptab, versus dir)
{
  if (dir == SX)
    *ptab = (*ptab)->sinistra;
  else if (dir == DX)
    *ptab = (*ptab)->dextra;
  else if (dir == DEORSUM)
    *ptab = (*ptab)->deorsum;
  else if (dir == SURSUM)
    *ptab = (*ptab)->sursum;
  return *ptab;
}
```

Listato gioca_tuki_esploraeritorna.c

```
/*_____________________________________________________________
 *
 * FILE: gioca_tuki_esploraeritorna.c
 */
#include "tuki5_modello.h"
#include "libagri.h"
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#define SCONOSCIUTO -2
#define GUINZAGLIO 5

char * dir(versus d)
{
  if(d == SX) return "SX";
  if(d == DX) return "DX";
  if(d == DEORSUM) return "DEORSUM";
  if(d == SURSUM) return "SURSUM";
  if(d == FIXO) return "FIXO";
  return "-1";
}

agri_Vertex agri_Vertices_Colligati[NNODI];

double euri(int start, int goal)
{
  int x1, x2, y1, y2;
  double d;
  x1 = agri_Vertices_Colligati[start].columna;
  y1 = agri_Vertices_Colligati[start].linea;
  x2 = agri_Vertices_Colligati[goal].columna;
  y2 = agri_Vertices_Colligati[goal].linea;

  d = (x1-x2)*(x1-x2)+(y1-y2)*(y1-y2);

  return sqrt(d);
}

int distanze[NNODI][NNODI];

double dist(int da_nodus, int a_nodus)
{
  int s = da_nodus;
  int g = a_nodus;

  return (double)distanze[s][g];
}

void stampa(agri_Colligationes_Colligatae g)
{
  FILE * f = fopen("grafi.csv","w+t");
  while(g)
    {
      fprintf(f," %d,%d,\n",
              g->colligatio.ab.index,
              g->colligatio.ad.index);

      g = g->next;
    }
  fclose(f);
}

bool oggetto_accessibile(oggetto s)
{
  if(s == 'J' || s == 'U' || s == 'V')
    return true;
  else
    return false;
}

typedef enum {ESPLORA, DECIDI, RITORNA} Modo;

direzione gioca_tuki(posizioni posi, oggetto **labx)
{
```

```
static agri_Verticum_Dispositio vert_disp;
static int nodi_percorsi = 0;
static int * percorso_fuga = 0;
static int * copia;
static Modo modo_gioco = ESPLORA;
static int mosse = 0;
mosse++;

labx[14][12]='A';
labx[14][13]='A';
labx[14][11]='A';
labx[14][14]='A';

static agri_Colligationes_Colligatae g = 0;

static int vertice_da = SCONOSCIUTO;
static int vertici_contati = SCONOSCIUTO;
static int direzione_arrivo = FERMO;
static int direzione_partenza = FERMO;
static int longitudo_colligatio = 0;
static int i_da, j_da;

static direzione ld = SINISTRA;

static bool init = false;
if(!init)
  {
    srand(time(0));
    init = true;
  }

int i = posi.tuki_y;
int j = posi.tuki_x;

longitudo_colligatio++;

oggetto vicino[4];
vicino[0] = labx[i][j-1]; //sinistra - left
vicino[1] = labx[i][j+1]; //destra - right
vicino[2] = labx[i-1][j]; //su - up
vicino[3] = labx[i+1][j]; //giu - down

oggetto s = vicino[0];
oggetto d = vicino[1];
oggetto a = vicino[2];
oggetto b = vicino[3];

int x = posi.tuki_x;
int y = posi.tuki_y;
int x_g[4];
int y_g[4];
x_g[0] = posi.blinky_x;
x_g[1] = posi.pinky_x;
x_g[2] = posi.inky_x;
x_g[3] = posi.clyde_x;

y_g[0] = posi.blinky_y;
y_g[1] = posi.pinky_y;
y_g[2] = posi.inky_y;
y_g[3] = posi.clyde_y;

char s_g = 0, d_g = 0, a_g = 0, b_g = 0;
for (int ig=0; ig<4; ig++)
  {
    s_g = s_g || ( x_g[ig] < x) && ( x - x_g[ig] <=2 )  && (y == y_g[ig]);
    s_g = s_g || ((x_g[ig] == x-1) &&  (y_g[ig] == y+1));
    s_g = s_g || ((x_g[ig] == x-1) &&  (y_g[ig] == y-1));
    d_g = d_g || ( x_g[ig] > x) && (x_g[ig] - x <= 2) && (y == y_g[ig]);
    d_g = d_g || ((x_g[ig] == x+1) &&  (y_g[ig] == y+1));
    d_g = d_g || ((x_g[ig] == x+1) &&  (y_g[ig] == y-1));
    a_g = a_g || (x == x_g[ig]) && ( y >  y_g[ig]) && ( y - y_g[ig] <=2);
    a_g = a_g || ((y == y_g[ig] + 1) && (x_g[ig] == x+1));
    a_g = a_g || ((y == y_g[ig] + 1) && (x_g[ig] == x-1));
    b_g = b_g || (x == x_g[ig]) && ( y <  y_g[ig]) && ( y_g[ig]- y <=2);
    b_g = b_g || ((y == y_g[ig] - 1) && (x_g[ig] == x+1));
    b_g = b_g || ((y == y_g[ig] - 1) && (x_g[ig] == x-1));
  }

int nd = 0;
for(int k=0; k<4; k++)
  nd += (1*oggetto_accessibile(vicino[k]));

fflush(stdout);
```

```
bool nodo_rilevato = false;

if(vertice_da == SCONOSCIUTO && nd>2)
  {
    fflush(stdout);
    vertice_da = 0;
    i_da = i;
    j_da = j;
    vertici_contati = 1;
    longitudo_colligatio = 0;
  }
else if(nd>2)
  {
    int vertice_a = agri_Verticem_quaero(g,i,j);
    if(vertice_a<0)
      {
        vertice_a = vertici_contati;
        vertici_contati++;
        nodi_percorsi++;
      }
    agri_Colligatio colligatio;
    agri_Vertex v_a, v_da;
    v_a = agri_Verticem_creo(vertice_a,i,j);
    v_da = agri_Verticem_creo(vertice_da,i_da,j_da);

    distanze[vertice_da][vertice_a] = longitudo_colligatio;

    colligatio.ad = v_a;
    colligatio.ab = v_da;
    colligatio.longitudo = longitudo_colligatio;
    colligatio.meta = direzione_arrivo;
    colligatio.discessus = direzione_partenza;
    agri_Colligationem_insero(&g, colligatio);
    nodo_rilevato = true;

    longitudo_colligatio = 0;
    vertice_da = vertice_a;
    i_da = i;
    j_da = j;

    stampa(g);
  }

bool disponibile = false;

bool aleatorio = false;

if(nodi_percorsi == GUINZAGLIO)
  modo_gioco = DECIDI;

if((s_g || d_g || a_g || b_g) && FUGA)
  {
    direzione esc[4];
    for(int i=0;i<4;i++) esc[i]=FERMO;
    int ki = 0; //direzioni buone, good directions
    if(copia) free(copia);
    copia = 0;
    modo_gioco = ESPLORA;
    nodi_percorsi = 0;

    if(oggetto_accessibile(s) && !s_g)
      {
        esc[ki] = SINISTRA;
        ki++;
      }
    if(oggetto_accessibile(a) && !a_g)
      {
        esc[ki] = SU;
        ki++;
      }
    if(oggetto_accessibile(d) && !d_g)
      {
        esc[ki] = DESTRA;
        ki++;
      }
    if(oggetto_accessibile(b) && !b_g)
      {
        esc[ki] = GIU;
        ki++;
      }
    if(ki == 0)
      {
```

```
        return FERMO;
      }
    ld = esc[rand()%ki];

    direzione_arrivo = ld;
    if(nodo_rilevato)
      direzione_partenza = ld;
    return ld;
  }

if(modo_gioco == ESPLORA)
  {
    while(!disponibile)
      {
        if(!oggetto_accessibile(s) && ld == SINISTRA)
          {
            ld = rand()%2;
            if(ld==0)
              ld = SU;
            else
              ld = GIU;
            aleatorio = true;
          }
        else if(!oggetto_accessibile(d) && ld == DESTRA)
          {
            ld = rand()%2;
            if(ld==0) ld = SU;
            else
              ld = GIU;
            aleatorio = true;
          }
        else if(!oggetto_accessibile(a) && ld == SU)
          {
            ld = rand()%2;
            if(ld==0) ld = SINISTRA;
            else
              ld = DESTRA;
            aleatorio = true;
          }
        else if(!oggetto_accessibile(b) && ld == GIU)
          {
            ld = rand()%2;
            if(ld==0) ld = SINISTRA;
            else
              ld = DESTRA;
            aleatorio = true;
          }
        else
          disponibile = true;
      }

    direzione_arrivo = ld;
    if(nodo_rilevato)
      direzione_partenza = ld;

    if(aleatorio) return ld;

    if(oggetto_accessibile(a) && ld !=SU && ld!=GIU)
      {
        int sv = rand()%10;
        if(sv>=5)
          ld = SU;
      }
    if(oggetto_accessibile(b) && ld !=GIU && ld!=SU)
      {
        int sv = rand()%10;
        if(sv>=5)
          ld = GIU;
      }
    if(oggetto_accessibile(s) && ld !=SINISTRA && ld!=DESTRA)
      {
        int sv = rand()%10;
        if(sv>=5)
          ld = SINISTRA;
      }
    if(oggetto_accessibile(d) && ld !=DESTRA && ld!=SINISTRA)
      {
        int sv = rand()%10;
        if(sv>=5)
          ld = DESTRA;
      }
    direzione_arrivo = ld;
```

```
    if(nodo_rilevato)
      direzione_partenza = ld;

    return ld;
  }

else if(modo_gioco == DECIDI)
  {
    nodi_percorsi = 0;
    int s = agri_muto(g,&vert_disp);

    percorso_fuga = agri_astar(vertice_da,0,vert_disp,&dist,&euri,vertici_contati);
    if(percorso_fuga == 0)
      {
        modo_gioco = ESPLORA;
        ld = FERMO;
        return ld;
      }
    copia = percorso_fuga;
    modo_gioco = RITORNA;
  }
if(modo_gioco == RITORNA)
  {
    if(nodo_rilevato == true)
      {
        int indice_nodo = *percorso_fuga;
        if(indice_nodo == -1)
          {
            nodi_percorsi = 0;
            modo_gioco = ESPLORA;
            free(copia);
            copia = 0;
            return rand()%4;
          }
        percorso_fuga++;

        if(vert_disp[vertice_da].ianua[SINISTRA] == indice_nodo)
          {
            ld = SINISTRA;
          }
        if(vert_disp[vertice_da].ianua[DESTRA] == indice_nodo)
          {
            ld = DESTRA;
          }
        if(vert_disp[vertice_da].ianua[SU] == indice_nodo)
          {
            ld = SU;
          }
        if(vert_disp[vertice_da].ianua[GIU] == indice_nodo)
          {
            ld = GIU;
          }
      }

    //SINISTRA
    if(ld == SINISTRA && oggetto_accessibile(s))
      return ld;
    if(ld == SINISTRA && oggetto_accessibile(a))
      return ld = SU;
    if(ld == SINISTRA && oggetto_accessibile(b))
      return ld = GIU;

    //DESTRA
    if(ld == DESTRA && oggetto_accessibile(d))
      return ld;
    if(ld == DESTRA && oggetto_accessibile(a))
      return ld = SU;
    if(ld == DESTRA && oggetto_accessibile(b))
      return ld = GIU;

    //SU
    if(ld == SU && oggetto_accessibile(a))
      return ld;
    if(ld == SU && oggetto_accessibile(s))
      return ld = SINISTRA;
    if(ld == SU && oggetto_accessibile(d))
      return ld = DESTRA;

    //GIU
    if(ld == GIU && oggetto_accessibile(b))
      return ld;
    if(ld == GIU && oggetto_accessibile(s))
      return ld = SINISTRA;
```

```
      if(ld == GIU && oggetto_accessibile(d))
        return ld = DESTRA;
    }
}
```

Listato gioca_tuki_generagrafo.c

```
/*_____________________________________________________________
 *
 * FILE: gioca_tuki_generagrafo.c
 */
#include "tuki5_modello.h"
#include "libagri.h"
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#define SCONOSCIUTO -2

char * dir(versus d)
{
  if(d == SX) return "SX";
  if(d == DX) return "DX";
  if(d == DEORSUM) return "DEORSUM";
  if(d == SURSUM) return "SURSUM";
  if(d == FIXO) return "FIXO";
  return "-1";
}

agri_Vertex agri_Vertices_Colligati[NNODI];
double euri(int start, int goal)
{
  int x1,x2,y1,y2;
  double d;
  x1=agri_Vertices_Colligati[start].columna;
  y1=agri_Vertices_Colligati[start].linea;
  x2=agri_Vertices_Colligati[goal].columna;
  y2=agri_Vertices_Colligati[goal].linea;

  d = (x1-x2)*(x1-x2)+(y1-y2)*(y1-y2);
  return sqrt(d);
}

int d[NNODI][NNODI];

double dist(int da_nodus, int a_nodus)
{
  int s = da_nodus;
  int g = a_nodus;

  return (double)d[s][g];
}

void stampa(agri_Colligationes_Colligatae g)
{
  FILE * f = fopen("grafi.csv","w+t");
  while(g)
    {
      fprintf(f," %d,%d,\n",
              g->colligatio.ab.index,
              g->colligatio.ad.index);
      g = g->next;
    }
  fclose(f);
}

bool oggetto_accessibile(oggetto s)
{
  if(s == 'J' || s == 'U' || s == 'V')
    return true;
  else
    return false;
}

direzione gioca_tuki(posizioni posi, oggetto **labx)
```

```
{
  static int mosse = 0;
  mosse++;

  labx[14][12]='A';
  labx[14][13]='A';
  labx[14][11]='A';
  labx[14][14]='A';

  static agri_Colligationes_Colligatae g = 0;
  static int vertice_da = SCONOSCIUTO;
  static int vertici_contati = SCONOSCIUTO;

  static int direzione_arrivo = FERMO;
  static int direzione_partenza = FERMO;

  static int longitudo_colligatio = 0;
  static int i_da, j_da;

  static direzione ld = SINISTRA;

  static bool init = false;
  if(!init)
    {
      srand(time(0));
      init = true;
    }

  int i = posi.tuki_y;
  int j = posi.tuki_x;

  longitudo_colligatio++;

  oggetto vicino[4];
  vicino[0] = labx[i][j-1]; //sinistra - left
  vicino[1] = labx[i][j+1]; //destra - right
  vicino[2] = labx[i-1][j]; //su - up
  vicino[3] = labx[i+1][j]; //giu - down

  int nd = 0;
  for(int k=0; k<4; k++)
    nd += (1*oggetto_accessibile(vicino[k]));

  fflush(stdout);

  bool nodo_rilevato = false;

  if(vertice_da == SCONOSCIUTO && nd>2)
    {
      fflush(stdout);
      vertice_da = 0;
      i_da = i;
      j_da = j;
      vertici_contati = 1;
      longitudo_colligatio = 0;
    }
  else if(nd>2)
    {
      int vertice_a = agri_Verticem_quaero(g,i,j);
      if(vertice_a<0)
        {
          vertice_a = vertici_contati;
          vertici_contati++;
        }
      agri_Colligatio colligatio;
      agri_Vertex v_a, v_da;
      v_a = agri_Verticem_creo(vertice_a,i,j);
      v_da = agri_Verticem_creo(vertice_da,i_da,j_da);

      d[vertice_da][vertice_a] = longitudo_colligatio;

      colligatio.ad = v_a;
      colligatio.ab = v_da;
      colligatio.longitudo = longitudo_colligatio;
      colligatio.meta = direzione_arrivo;
      colligatio.discessus = direzione_partenza;
      agri_Colligationem_insero(&g, colligatio);
      nodo_rilevato = true;

      longitudo_colligatio = 0;
      vertice_da = vertice_a;
      i_da = i;
      j_da = j;
```

```
    stampa(g);
  }

oggetto s = vicino[0];
oggetto d = vicino[1];
oggetto a = vicino[2];
oggetto b = vicino[3];

bool disponibile = false;
bool aleatorio = false;

while(!disponibile)
  {
    if(!oggetto_accessibile(s) && ld == SINISTRA)
      {
        ld = rand()%2;
        if(ld==0)
          ld = SU;
        else
          ld = GIU;
        aleatorio = true;
      }
    else if(!oggetto_accessibile(d) && ld == DESTRA)
      {
        ld = rand()%2;
        if(ld==0) ld = SU;
        else
          ld = GIU;
        aleatorio = true;
      }
    else if(!oggetto_accessibile(a) && ld == SU)
      {
        ld = rand()%2;
        if(ld==0) ld = SINISTRA;
        else
          ld = DESTRA;
        aleatorio = true;
      }
    else if(!oggetto_accessibile(b) && ld == GIU)
      {
        ld = rand()%2;
        if(ld==0) ld = SINISTRA;
        else
          ld = DESTRA;
        aleatorio = true;
      }
    else
      disponibile = true;
  }

direzione_arrivo = ld;
if(nodo_rilevato)
  direzione_partenza = ld;

if(aleatorio) return ld;

if(oggetto_accessibile(a) && ld !=SU && ld!=GIU)
  {
    int sv = rand()%10;
    if(sv>=5)
      ld = SU;
  }
if(oggetto_accessibile(b) && ld !=GIU && ld!=SU)
  {
    int sv = rand()%10;
    if(sv>=5)
      ld = GIU;
  }
if(oggetto_accessibile(s) && ld !=SINISTRA && ld!=DESTRA)
  {
    int sv = rand()%10;
    if(sv>=5)
      ld = SINISTRA;
  }
if(oggetto_accessibile(d) && ld !=DESTRA && ld!=SINISTRA)
  {
    int sv = rand()%10;
    if(sv>=5)
      ld = DESTRA;
  }

direzione_arrivo = ld;
```

```
  if(nodo_rilevato)
    direzione_partenza = ld;
  return ld;
}
```

Listato gioca_tuki_percorso.c

```
/*_____________________________________________________________
 *
 * FILE: gioca_tuki_percorso.c
 */
#include "tuki5_modello.h"
#include "libagri.h"
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>

#define SCONOSCIUTO -2
#define GUINZAGLIO 10

#define NODI_LAB_POT 44

static agri_Vertex grafo[NODI_LAB_POT];

int * cammino;
char * dir(versus d)
{
  if(d == SX) return "SX";
  if(d == DX) return "DX";
  if(d == DEORSUM) return "DEORSUM";
  if(d == SURSUM) return "SURSUM";
  if(d == FIXO) return "FIXO";
  return "-1";
}

agri_Vertex agri_Vertices_Colligati[NODI_LAB_POT];
double euri(int start, int goal)
{
  int x1,x2,y1,y2;
  double d;
  x1=agri_Vertices_Colligati[start].columna;
  y1=agri_Vertices_Colligati[start].linea;
  x2=agri_Vertices_Colligati[goal].columna;
  y2=agri_Vertices_Colligati[goal].linea;

  d = (x1-x2)*(x1-x2)+(y1-y2)*(y1-y2);
  return sqrt(d);
}

bool oggetto_accessibile(oggetto s)
{
  if(s == 'J' || s == 'U' || s == 'V')
    return true;
  else
    return false;
}

void collega_tuki_nodi()
{
  static int cam[]=
    {
      3,10,38,2,34,0,3,2,3,0,35,5,4,3,4,12,13,
      7,6,5,6,36,1,8,7,8,1,37,9,39,11,39,9,8,11,17,
      16,13,12,15,14,17,23,22,19,16,19,15,
      21,40,20,14,20,40,30,42,32,33,43,31,41,
      23,29,31,29,28,33,28,27,22,27,26,21,26,
      25,32,25,24,20,24,30
    };

  cammino = cam;

  grafo[0].linea = 3;
  grafo[0].columna = 6;
  grafo[0].index = 0;
```

```
grafo[0].ianua[SINISTRA] = 34;
grafo[0].ianua[DESTRA] = 35;
grafo[0].ianua[SU] = -1;
grafo[0].ianua[GIU] = 3;

grafo[1].linea = 3;
grafo[1].columna = 21;
grafo[1].index = 1;
grafo[1].ianua[SINISTRA] = 36;
grafo[1].ianua[DESTRA] = 37;
grafo[1].ianua[SU] = -1;
grafo[1].ianua[GIU] = 8;

grafo[2].linea = 7;
grafo[2].columna = 1;
grafo[2].index = 2;
grafo[2].ianua[SINISTRA] = -1;
grafo[2].ianua[DESTRA] = 3;
grafo[2].ianua[SU] = 34;
grafo[2].ianua[GIU] = 38;

grafo[3].linea = 7;
grafo[3].columna = 6;
grafo[3].index = 3;
grafo[3].ianua[SINISTRA] = 2;
grafo[3].ianua[DESTRA] = 4;
grafo[3].ianua[SU] = 0;
grafo[3].ianua[GIU] = 10;

grafo[4].linea = 7;
grafo[4].columna = 9;
grafo[4].index = 4;
grafo[4].ianua[SINISTRA] = 3;
grafo[4].ianua[DESTRA] = 5;
grafo[4].ianua[SU] = -1;
grafo[4].ianua[GIU] = 12;

grafo[5].linea = 7;
grafo[5].columna = 12;
grafo[5].index = 5;
grafo[5].ianua[SINISTRA] = 4;
grafo[5].ianua[DESTRA] = 6;
grafo[5].ianua[SU] = 35;
grafo[5].ianua[GIU] = -1;

grafo[6].linea = 7;
grafo[6].columna = 15;
grafo[6].index = 6;
grafo[6].ianua[SINISTRA] = 5;
grafo[6].ianua[DESTRA] = 7;
grafo[6].ianua[SU] = 36;
grafo[6].ianua[GIU] = -1;

grafo[7].linea = 7;
grafo[7].columna = 18;
grafo[7].index = 7;
grafo[7].ianua[SINISTRA] = 6;
grafo[7].ianua[DESTRA] = 8;
grafo[7].ianua[SU] = -1;
grafo[7].ianua[GIU] = 13;

grafo[8].linea = 7;
grafo[8].columna = 21;
grafo[8].index = 8;
grafo[8].ianua[SINISTRA] = 7;
grafo[8].ianua[DESTRA] = 9;
grafo[8].ianua[SU] = 1;
grafo[8].ianua[GIU] = 11;

grafo[9].linea = 7;
grafo[9].columna = 26;
grafo[9].index = 9;
grafo[9].ianua[SINISTRA] = 8;
grafo[9].ianua[DESTRA] = -1;
grafo[9].ianua[SU] = 37;
grafo[9].ianua[GIU] = 39;

grafo[10].linea = 10;
grafo[10].columna = 6;
grafo[10].index = 10;
grafo[10].ianua[SINISTRA] = 38;
grafo[10].ianua[DESTRA] = -1;
grafo[10].ianua[SU] = 3;
```

```
grafo[10].ianua[GIU] = 14;

grafo[11].linea = 10;
grafo[11].columna = 21;
grafo[11].index = 11;
grafo[11].ianua[SINISTRA] = -1;
grafo[11].ianua[DESTRA] = 39;
grafo[11].ianua[SU] = 8;
grafo[11].ianua[GIU] = 17;

grafo[12].linea = 13;
grafo[12].columna = 12;
grafo[12].index = 12;
grafo[12].ianua[SINISTRA] = 15;
grafo[12].ianua[DESTRA] = 13;
grafo[12].ianua[SU] = 4;
grafo[12].ianua[GIU] = -1;

grafo[13].linea = 13;
grafo[13].columna = 15;
grafo[13].index = 13;
grafo[13].ianua[SINISTRA] = 12;
grafo[13].ianua[DESTRA] = 16;
grafo[13].ianua[SU] = 7;
grafo[13].ianua[GIU] = -1;

grafo[14].linea = 16;
grafo[14].columna = 6;
grafo[14].index = 14;
grafo[14].ianua[SINISTRA] = -1;
grafo[14].ianua[DESTRA] = 15;
grafo[14].ianua[SU] = 10;
grafo[14].ianua[GIU] = 20;

grafo[15].linea = 16;
grafo[15].columna = 9;
grafo[15].index = 15;
grafo[15].ianua[SINISTRA] = 14;
grafo[15].ianua[DESTRA] = -1;
grafo[15].ianua[SU] = 12;
grafo[15].ianua[GIU] = 18;

grafo[16].linea = 16;
grafo[16].columna = 18;
grafo[16].index = 16;
grafo[16].ianua[SINISTRA] = -1;
grafo[16].ianua[DESTRA] = 17;
grafo[16].ianua[SU] = 13;
grafo[16].ianua[GIU] = 19;

grafo[17].linea = 16;
grafo[17].columna = 21;
grafo[17].index = 17;
grafo[17].ianua[SINISTRA] = 16;
grafo[17].ianua[DESTRA] = -1;
grafo[17].ianua[SU] = 11;
grafo[17].ianua[GIU] = 23;

grafo[18].linea = 19;
grafo[18].columna = 9;
grafo[18].index = 18;
grafo[18].ianua[SINISTRA] = -1;
grafo[18].ianua[DESTRA] = 19;
grafo[18].ianua[SU] = 15;
grafo[18].ianua[GIU] = 21;

grafo[19].linea = 19;
grafo[19].columna = 18;
grafo[19].index = 19;
grafo[19].ianua[SINISTRA] = 18;
grafo[19].ianua[DESTRA] = -1;
grafo[19].ianua[SU] = 16;
grafo[19].ianua[GIU] = 22;

grafo[20].linea = 22;
grafo[20].columna = 6;
grafo[20].index = 20;
grafo[20].ianua[SINISTRA] = 40;
grafo[20].ianua[DESTRA] = 21;
grafo[20].ianua[SU] = 14;
grafo[20].ianua[GIU] = 24;

 grafo[21].linea = 22;
```

```
grafo[21].columna = 9;
grafo[21].index = 21;
grafo[21].ianua[SINISTRA] = 20;
grafo[21].ianua[DESTRA] = 26;
grafo[21].ianua[SU] = 18;
grafo[21].ianua[GIU] = -1;

grafo[22].linea = 22;
grafo[22].columna = 18;
grafo[22].index = 22;
grafo[22].ianua[SINISTRA] = 27;
grafo[22].ianua[DESTRA] = 23;
grafo[22].ianua[SU] = 19;
grafo[22].ianua[GIU] = -1;

grafo[23].linea = 22;
grafo[23].columna = 21;
grafo[23].index = 23;
grafo[23].ianua[SINISTRA] = 22;
grafo[23].ianua[DESTRA] = 41;
grafo[23].ianua[SU] = 17;
grafo[23].ianua[GIU] = 29;

grafo[24].linea = 25;
grafo[24].columna = 6;
grafo[24].index = 24;
grafo[24].ianua[SINISTRA] = -1;
grafo[24].ianua[DESTRA] = 25;
grafo[24].ianua[SU] = 20;
grafo[24].ianua[GIU] = 30;

grafo[25].linea = 25;
grafo[25].columna = 9;
grafo[25].index = 25;
grafo[25].ianua[SINISTRA] = 24;
grafo[25].ianua[DESTRA] = 26;
grafo[25].ianua[SU] = -1;
grafo[25].ianua[GIU] = 32;

grafo[26].linea = 25;
grafo[26].columna = 12;
grafo[26].index = 26;
grafo[26].ianua[SINISTRA] = 25;
grafo[26].ianua[DESTRA] = 27;
grafo[26].ianua[SU] = 21;
grafo[26].ianua[GIU] = -1;

grafo[27].linea = 25;
grafo[27].columna = 15;
grafo[27].index = 27;
grafo[27].ianua[SINISTRA] = 26;
grafo[27].ianua[DESTRA] = 28;
grafo[27].ianua[SU] = 22;
grafo[27].ianua[GIU] = -1;

grafo[28].linea = 25;
grafo[28].columna = 18;
grafo[28].index = 28;
grafo[28].ianua[SINISTRA] = 27;
grafo[28].ianua[DESTRA] = 29;
grafo[28].ianua[SU] = -1;
grafo[28].ianua[GIU] = 33;

grafo[29].linea = 25;
grafo[29].columna = 21;
grafo[29].index = 29;
grafo[29].ianua[SINISTRA] = 28;
grafo[29].ianua[DESTRA] = -1;
grafo[29].ianua[SU] = 23;
grafo[29].ianua[GIU] = 31;

grafo[30].linea = 28;
grafo[30].columna = 3;
grafo[30].index = 30;
grafo[30].ianua[SINISTRA] = 42;
grafo[30].ianua[DESTRA] = 24;
grafo[30].ianua[SU] = 40;
grafo[30].ianua[GIU] = -1;

grafo[31].linea = 28;
grafo[31].columna = 24;
grafo[31].index = 31;
grafo[31].ianua[SINISTRA] = 29;
```

```
grafo[31].ianua[DESTRA] = 43;
grafo[31].ianua[SU] = 41;
grafo[31].ianua[GIU] = -1;

grafo[32].linea = 31;
grafo[32].columna = 12;
grafo[32].index = 32;
grafo[32].ianua[SINISTRA] = 42;
grafo[32].ianua[DESTRA] = 33;
grafo[32].ianua[SU] = 25;
grafo[32].ianua[GIU] = -1;

grafo[33].linea = 31;
grafo[33].columna = 15;
grafo[33].index = 33;
grafo[33].ianua[SINISTRA] = 32;
grafo[33].ianua[DESTRA] = 43;
grafo[33].ianua[SU] = 28;
grafo[33].ianua[GIU] = -1;

grafo[34].linea = 3;
grafo[34].columna = 1;
grafo[34].index = 34;
grafo[34].ianua[SINISTRA] = -1;
grafo[34].ianua[DESTRA] = 0;
grafo[34].ianua[SU] = -1;
grafo[34].ianua[GIU] = 2;

grafo[35].linea = 3;
grafo[35].columna = 12;
grafo[35].index = 35;
grafo[35].ianua[SINISTRA] = 0;
grafo[35].ianua[DESTRA] = -1;
grafo[35].ianua[SU] = -1;
grafo[35].ianua[GIU] = 5;

grafo[36].linea = 3;
grafo[36].columna = 15;
grafo[36].index = 36;
grafo[36].ianua[SINISTRA] = -1;
grafo[36].ianua[DESTRA] = 1;
grafo[36].ianua[SU] = -1;
grafo[36].ianua[GIU] = 6;

grafo[37].linea = 3;
grafo[37].columna = 26;
grafo[37].index = 37;
grafo[37].ianua[SINISTRA] = 1;
grafo[37].ianua[DESTRA] = -1;
grafo[37].ianua[SU] = -1;
grafo[37].ianua[GIU] = 9;

grafo[38].linea = 10;
grafo[38].columna = 1;
grafo[38].index = 38;
grafo[38].ianua[SINISTRA] = -1;
grafo[38].ianua[DESTRA] = 10;
grafo[38].ianua[SU] = 2;
grafo[38].ianua[GIU] = -1;

grafo[39].linea = 10;
grafo[39].columna = 26;
grafo[39].index = 39;
grafo[39].ianua[SINISTRA] = 11;
grafo[39].ianua[DESTRA] = -1;
grafo[39].ianua[SU] = 9;
grafo[39].ianua[GIU] = -1;

grafo[40].linea = 22;
grafo[40].columna = 1;
grafo[40].index = 40;
grafo[40].ianua[SINISTRA] = -1;
grafo[40].ianua[DESTRA] = 20;
grafo[40].ianua[SU] = -1;
grafo[40].ianua[GIU] = 30;

grafo[41].linea = 22;
grafo[41].columna = 26;
grafo[41].index = 41;
grafo[41].ianua[SINISTRA] = 23;
grafo[41].ianua[DESTRA] = -1;
grafo[41].ianua[SU] = -1;
grafo[41].ianua[GIU] = 31;
```

```
  grafo[42].linea = 31;
  grafo[42].columna = 1;
  grafo[42].index = 42;
  grafo[42].ianua[SINISTRA] = -1;
  grafo[42].ianua[DESTRA] = 32;
  grafo[42].ianua[SU] = 30;
  grafo[42].ianua[GIU] = -1;

  grafo[43].linea = 31;
  grafo[43].columna = 26;
  grafo[43].index = 43;
  grafo[43].ianua[SINISTRA] = 33;
  grafo[43].ianua[DESTRA] = -1;
  grafo[43].ianua[SU] = 31;
  grafo[43].ianua[GIU] = -1;
}

double distanza_esatta(int da_nodo, int a_nodo)
{
  int s = da_nodo;
  int g = a_nodo;
  static int distanze[NODI_LAB_POT][NODI_LAB_POT];
  static int init = 0;
  if(!init)
    {
      for(int i=0; i<NODI_LAB_POT;i++)
        for(int j=0; j<NODI_LAB_POT;j++)
          distanze[i][j]=INFINITO;

      distanze[0][2]=8;
      distanze[0][3]=3;
      distanze[0][5]=9;
      distanze[2][0]=8;
      distanze[3][0]=3;
      distanze[5][0]=9;

      distanze[1][6]=9;
      distanze[1][8]=3;
      distanze[1][9]=8;
      distanze[6][1]=9;
      distanze[8][1]=3;
      distanze[9][1]=8;

      distanze[2][3]=4;
      distanze[2][10]=7;
      distanze[3][2]=4;
      distanze[10][2]=7;

      distanze[3][4]=2;
      distanze[3][10]=2;
      distanze[4][3]=2;
      distanze[10][3]=2;

      distanze[4][5]=2;
      distanze[4][12]=8;
      distanze[5][4]=2;
      distanze[12][4]=8;

      distanze[5][6]=2;
      distanze[6][5]=2;

      distanze[6][7]=2;
      distanze[7][6]=2;

      distanze[7][8]=2;
      distanze[7][13]=8;
      distanze[8][7]=2;
      distanze[13][7]=8;

      distanze[8][9]=4;
      distanze[8][11]=2;
      distanze[9][8]=4;
      distanze[11][8]=2;

      distanze[9][11]=7;
      distanze[11][9]=7;

      distanze[10][14]=5;
      distanze[14][10]=5;

      distanze[11][17]=5;
      distanze[17][11]=5;
```

```
distanze[12][13]=2;
distanze[12][15]=5;
distanze[13][12]=2;
distanze[15][12]=5;

distanze[13][16]=5;
distanze[16][13]=5;

distanze[14][15]=2;
distanze[14][20]=5;
distanze[15][14]=2;
distanze[20][14]=5;

distanze[14][17]=6;
distanze[17][14]=6;

distanze[15][18]=2;
distanze[18][15]=2;

distanze[16][17]=2;
distanze[16][19]=2;
distanze[17][16]=2;
distanze[19][16]=2;

distanze[17][23]=5;
distanze[23][17]=5;

distanze[18][19]=8;
distanze[18][21]=2;
distanze[19][18]=8;
distanze[21][18]=2;

distanze[19][22]=2;
distanze[22][19]=2;

distanze[20][30]=12;
distanze[20][24]=2;
distanze[20][21]=2;
distanze[30][20]=12;
distanze[24][20]=2;
distanze[21][20]=2;

distanze[21][26]=5;
distanze[26][21]=5;

distanze[22][27]=5;
distanze[22][23]=2;
distanze[27][22]=5;
distanze[23][22]=2;

distanze[23][29]=2;
distanze[23][31]=12;
distanze[29][23]=2;
distanze[31][23]=12;

distanze[24][30]=5;
distanze[24][25]=2;
distanze[30][24]=5;
distanze[25][24]=2;

distanze[25][26]=2;
distanze[25][32]=8;
distanze[26][25]=2;
distanze[32][25]=8;

distanze[26][27]=2;
distanze[27][26]=2;

distanze[27][28]=2;
distanze[28][27]=2;

distanze[28][33]=8;
distanze[28][29]=2;
distanze[33][28]=8;
distanze[29][28]=2;

distanze[29][31]=5;
distanze[31][29]=5;

distanze[30][32]=15;
distanze[32][30]=15;
```

```
      distanze[31][33]=15;
      distanze[33][31]=15;

      distanze[32][33]=2;
      distanze[33][32]=2;

      distanze[34][0]=4;
      distanze[0][34]=4;

      distanze[34][2]=3;
      distanze[2][34]=3;

      distanze[35][0]=5;
      distanze[0][35]=5;

      distanze[35][5]=3;
      distanze[5][35]=3;

      distanze[36][1]=5;
      distanze[1][36]=5;

      distanze[36][6]=3;
      distanze[6][36]=3;

      distanze[37][1]=4;
      distanze[1][37]=4;

      distanze[37][9]=3;
      distanze[9][37]=3;

      distanze[38][2]=2;
      distanze[2][38]=2;

      distanze[38][10]=4;
      distanze[10][38]=4;

      distanze[39][9]=2;
      distanze[9][39]=2;

      distanze[39][11]=4;
      distanze[11][39]=4;

      distanze[40][20]=4;
      distanze[20][40]=4;

      distanze[40][30]=7;
      distanze[30][40]=7;

      distanze[41][23]=4;
      distanze[23][41]=4;

      distanze[41][31]=7;
      distanze[31][41]=7;

      distanze[42][30]=4;
      distanze[30][42]=4;

      distanze[42][32]=10;
      distanze[32][42]=10;

      distanze[43][31]=4;
      distanze[31][43]=4;

      distanze[43][33]=10;
      distanze[33][43]=10;

      init=1;
    }
  return (double)distanze[s][g];
}

int indice_tuki_nodo_cella(int riga, int col)
{
  for(int i=0; i<NODI_LAB_POT;i++)
    {
      if(grafo[i].columna == col && grafo[i].linea == riga)
        return grafo[i].index;
    }
  return -1;
}

typedef enum {ESPLORA, CALCOLA, NAVIGA} Modo;
```

```
direzione gioca_tuki(posizioni posi, oggetto **labx)
{
  static int * percorso_fuga = 0;
  static int * copia;
  static Modo modo_gioco = ESPLORA;
  static int indice_array_vertice_bersaglio = 0;
  static bool init = false;
  static direzione ld = SINISTRA;
  int vertice_corrente = -1;

  labx[14][12]='A';
  labx[14][13]='A';
  labx[14][11]='A';
  labx[14][14]='A';

  if(!init)
    {
      collega_tuki_nodi();

      struct timeval time;
      gettimeofday(&time,NULL);
      srand((time.tv_sec * 1000) + (time.tv_usec / 1000));
      init = true;
    }

  int i = posi.tuki_y;
  int j = posi.tuki_x;

  oggetto vicino[4];
  vicino[0] = labx[i][j-1]; //sinistra - left
  vicino[1] = labx[i][j+1]; //destra - right
  vicino[2] = labx[i-1][j]; //su - up
  vicino[3] = labx[i+1][j]; //giu - down

  oggetto s = vicino[0];
  oggetto d = vicino[1];
  oggetto a = vicino[2];
  oggetto b = vicino[3];

  int x = posi.tuki_x;
  int y = posi.tuki_y;
  int x_g[4];
  int y_g[4];
  x_g[0] = posi.blinky_x;
  x_g[1] = posi.pinky_x;
  x_g[2] = posi.inky_x;
  x_g[3] = posi.clyde_x;

  y_g[0] = posi.blinky_y;
  y_g[1] = posi.pinky_y;
  y_g[2] = posi.inky_y;
  y_g[3] = posi.clyde_y;

  char s_g = 0, d_g = 0,a_g = 0,b_g = 0;
  for (int ig = 0; ig<4; ig++)
    {
      s_g = s_g || ( x_g[ig] < x) && ( x - x_g[ig] <=2 )  && (y == y_g[ig]);
      s_g = s_g || ((x_g[ig] == x-1) &&  (y_g[ig] == y+1));
      s_g = s_g || ((x_g[ig] == x-1) &&  (y_g[ig] == y-1));
      d_g = d_g || ( x_g[ig] > x) && (x_g[ig] - x <= 2) && (y == y_g[ig]);
      d_g = d_g || ((x_g[ig] == x+1) &&  (y_g[ig] == y+1));
      d_g = d_g || ((x_g[ig] == x+1) &&  (y_g[ig] == y-1));
      a_g = a_g || (x == x_g[ig]) && ( y >  y_g[ig]) && ( y - y_g[ig] <=2);
      a_g = a_g || ((y == y_g[ig] + 1) && (x_g[ig] == x+1));
      a_g = a_g || ((y == y_g[ig] + 1) && (x_g[ig] == x-1));
      b_g = b_g || (x == x_g[ig]) && ( y <  y_g[ig]) && ( y_g[ig]- y <=2);
      b_g = b_g || ((y == y_g[ig] - 1) && (x_g[ig] == x+1));
      b_g = b_g || ((y == y_g[ig] - 1) && (x_g[ig] == x-1));
    }

  int nd = 0;
  for(int k=0; k<4; k++)
    nd += (1*oggetto_accessibile(vicino[k]));

  fflush(stdout);

  bool nodo_rilevato = false;
  if(nd>=2)
    {
      vertice_corrente = indice_tuki_nodo_cella(i,j);
      if(vertice_corrente>=0)
        {
          nodo_rilevato = true;
```

```
        if(modo_gioco == ESPLORA)
          modo_gioco = CALCOLA;
      }
  }

bool disponibile = false;
bool aleatorio = false;

if((s_g || d_g || a_g || b_g) && FUGA)
  {
    direzione esc[4];
    for(int i=0;i<4;i++) esc[i]=FERMO;
    int ki = 0; //direzioni buone, good directions
    if(copia) free(copia);
    copia = 0;
    modo_gioco = ESPLORA;

    if(oggetto_accessibile(s) && !s_g)
      {
        esc[ki] = SINISTRA;
        ki++;
      }
    if(oggetto_accessibile(a) && !a_g)
      {
        esc[ki] = SU;
        ki++;
      }
    if(oggetto_accessibile(d) && !d_g)
      {
        esc[ki] = DESTRA;
        ki++;
      }
    if(oggetto_accessibile(b) && !b_g)
      {
        esc[ki] = GIU;
        ki++;
      }
    if(ki == 0) return FERMO;

    ld = esc[rand()%ki];

    return ld;
  }

if(modo_gioco == ESPLORA)
  {
    while(!disponibile)
      {
        if(!oggetto_accessibile(s) && ld == SINISTRA)
          {
            ld = rand()%2;
            if(ld==0)
              ld = SU;
            else
              ld = GIU;
            aleatorio = true;
          }
        else if(!oggetto_accessibile(d) && ld == DESTRA)
          {
            ld = rand()%2;
            if(ld==0) ld = SU;
            else
              ld = GIU;
            aleatorio = true;
          }
        else if(!oggetto_accessibile(a) && ld == SU)
          {
            ld = rand()%2;
            if(ld==0) ld = SINISTRA;
            else
              ld = DESTRA;
            aleatorio = true;
          }
        else if(!oggetto_accessibile(b) && ld == GIU)
          {
            ld = rand()%2;
            if(ld==0) ld = SINISTRA;
            else
              ld = DESTRA;
            aleatorio = true;
          }
        else disponibile = true;
      }
```

```
    if(aleatorio) return ld;

    if(oggetto_accessibile(a) && ld !=SU && ld!=GIU)
      {
        int sv = rand()%10;
        if(sv>=5)
          ld = SU;
      }
    if(oggetto_accessibile(b) && ld !=GIU && ld!=SU)
      {
        int sv = rand()%10;
        if(sv>=5)
          ld = GIU;
      }
    if(oggetto_accessibile(s) && ld !=SINISTRA && ld!=DESTRA)
      {
        int sv = rand()%10;
        if(sv>=5)
          ld = SINISTRA;
      }
    if(oggetto_accessibile(d) && ld !=DESTRA && ld!=SINISTRA)
      {
        int sv = rand()%10;
        if(sv>=5)
          ld = DESTRA;
      }

    return ld;
  }
else if(modo_gioco == CALCOLA)
  {
    if( vertice_corrente ==  cammino[indice_array_vertice_bersaglio])
      {
        indice_array_vertice_bersaglio++;
      }

    percorso_fuga = agri_astar
      (vertice_corrente,
       cammino[indice_array_vertice_bersaglio],
       grafo,&distanza_esatta,&euri,NODI_LAB_POT);

    if(percorso_fuga == 0)
      {
        modo_gioco = ESPLORA;
        ld = FERMO;
        return ld;
      }

    copia = percorso_fuga;
    modo_gioco = NAVIGA;
  }

if(modo_gioco == NAVIGA)
  {
    if(nodo_rilevato == true)
      {
        int indice_nodo = *percorso_fuga;
        if(indice_nodo == -1 ||(vertice_corrente ==cammino[indice_array_vertice_bersagli
          {
            indice_array_vertice_bersaglio++;
            modo_gioco = ESPLORA;
            free(copia);
            copia = 0;
            return FERMO;
          }
        percorso_fuga++;

        if(grafo[vertice_corrente].ianua[SINISTRA] == indice_nodo)
          {
            ld = SINISTRA;
          }
        if(grafo[vertice_corrente].ianua[DESTRA] == indice_nodo)
          {
            ld = DESTRA;
          }
        if(grafo[vertice_corrente].ianua[SU] == indice_nodo)
          {
            ld = SU;
          }
        if(grafo[vertice_corrente].ianua[GIU] == indice_nodo)
          {
            ld = GIU;
```

```
          }
        }

      //SINISTRA - LEFT
      if(ld == SINISTRA && oggetto_accessibile(s))
        return ld;
      if(ld == SINISTRA && oggetto_accessibile(a))
        return ld = SU;
      if(ld == SINISTRA && oggetto_accessibile(b))
        return ld = GIU;

      //DESTRA - RIGHT
      if(ld == DESTRA && oggetto_accessibile(d))
        return ld;
      if(ld == DESTRA && oggetto_accessibile(a))
        return ld = SU;
      if(ld == DESTRA && oggetto_accessibile(b))
        return ld = GIU;

      //SU - UP
      if(ld == SU && oggetto_accessibile(a))
        return ld;
      if(ld == SU && oggetto_accessibile(s))
        return ld = SINISTRA;
      if(ld == SU && oggetto_accessibile(d))
        return ld = DESTRA;

      //GIU - DOWN
      if(ld == GIU && oggetto_accessibile(b))
        return ld;
      if(ld == GIU && oggetto_accessibile(s))
        return ld = SINISTRA;
      if(ld == GIU && oggetto_accessibile(d))
        return ld = DESTRA;
    }
}
```

Listato gioca_tuki_pesato.c

```
/*_____________________________________________________________
 *
 * FILE: gioca_tuki_pesato.c
 */
#include "tuki5_modello.h"
#include "libagri.h"
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>

#define SCONOSCIUTO -2
#define GUINZAGLIO 10
#define NODI_LAB_POT 600

posizioni g_posi;
int n_vertici;
static agri_Vertex grafo[NODI_LAB_POT];
agri_Vertex agri_Vertices_Colligati[NODI_LAB_POT];


int evita_casa_fantasmi(int vertice)
{
  int r = grafo[vertice].linea;
  int c = grafo[vertice].columna;
  if(r>=15 && r<=17 && c>=11 && c<=16)
    return 0;

  return vertice;
}

int trova_vertice(int riga, int colonna)
{
  for(int i = 0; i<NODI_LAB_POT;i++)
    {
      if(grafo[i].linea == riga && grafo[i].columna == colonna)
```

```
        return grafo[i].index;
    }
  return -1;
}

bool oggetto_accessibile(oggetto s)
{
  if(s == 'J' || s == 'U' || s == 'V')
    return true;
  else
    return false;
}

bool area_accessibile(int r, int c)
{
  if (r<3) return false;
  if(r>32) return false;
  if( r>=12 && r<=14 && (c<=4 || c>= 23)) return false;
  if( r>=18 && r<=20 && (c<=4 || c>= 23)) return false;
  return true;
}

double euri(int start, int goal)
{
  if(start<0)
    {
      exit(1);
    }
  int x = g_posi.tuki_x;
  int y = g_posi.tuki_y;
  int x_g[4];
  int y_g[4];
  x_g[0] = g_posi.blinky_x;
  x_g[1] = g_posi.pinky_x;
  x_g[2] = g_posi.inky_x;
  x_g[3] = g_posi.clyde_x;

  y_g[0] = g_posi.blinky_y;
  y_g[1] = g_posi.pinky_y;
  y_g[2] = g_posi.inky_y;
  y_g[3] = g_posi.clyde_y;

  int peso_g[]={600,600,600,600};
  int x1,x2,y1,y2;
  double d;
  x1 = agri_Vertices_Colligati[start].columna;
  y1 = agri_Vertices_Colligati[start].linea;
  x2 = agri_Vertices_Colligati[goal].columna;
  y2 = agri_Vertices_Colligati[goal].linea;

  d = (x1-x2)*(x1-x2)+(y1-y2)*(y1-y2);

  for(int i = 0; i<4; i++)
    {
      int v = trova_vertice(y_g[i], x_g[i]);
      if(grafo[v].ianua[SINISTRA] == start ||
         grafo[v].ianua[DESTRA] == start ||
         grafo[v].ianua[SU] == start ||
         grafo[v].ianua[GIU] == start||
         grafo[v].index == start
         )
        {
          return peso_g[i];
        }
    }
  return(sqrt(d));
}

void collega_tuki_nodi(oggetto **labx)
{
  int i_aux,j_aux;
  int k = 0;
  /* Genera i vertici del grafo */
  for(int i = 3; i<ALTEZZA-3; i++)
    {
      for(int j = 1; j<LARGHEZZA-1; j++)
        {
          if( oggetto_accessibile(labx[i][j]) && area_accessibile(i,j))
            {
              grafo[k].linea = i;
              grafo[k].columna = j;
              grafo[k].index = k;
              k++;
```

```
            }
        }
    }

  n_vertici = k;
  /* Collega i vertici del grafo */
  for(int k = 0; k<NODI_LAB_POT; k++)
    {
      int r = grafo[k].linea;
      int c = grafo[k].columna;
      if(c>0)
        grafo[k].ianua[SINISTRA] =  trova_vertice(r, c-1);
      if(c<LARGHEZZA - 1)
        grafo[k].ianua[DESTRA] = trova_vertice(r, c+1);
      if(r>0)
        grafo[k].ianua[SU] = trova_vertice(r-1, c);
      if(r<ALTEZZA-1)
        grafo[k].ianua[GIU] = trova_vertice(r+1, c);
    }
}

double distanza_esatta(int da_nodo, int a_nodo)
{
  int s = da_nodo;
  int g = a_nodo;
  static int distanze[NODI_LAB_POT][NODI_LAB_POT];
  static int init = 0;
  if(!init)
    {
      for(int i=0; i<NODI_LAB_POT;i++)
        for(int j=0; j<NODI_LAB_POT;j++)
          {
            distanze[i][j]=INFINITO;

            if(grafo[da_nodo].ianua[SINISTRA] == a_nodo ||
               grafo[da_nodo].ianua[DESTRA] == a_nodo ||
               grafo[da_nodo].ianua[SU] == a_nodo ||
               grafo[da_nodo].ianua[GIU] == a_nodo)
              distanze[i][j] = 1;
          }
      init = 1;
    }
  return (double)distanze[s][g];
}

direzione gioca_tuki(posizioni posi, oggetto **labx)
{
  static int * percorso_fuga = 0;
  static int vertice_goal = -1;
  static int init = 0;
  g_posi = posi;
  int vertice_corrente = -1;

  if(!init)
    {
      collega_tuki_nodi(labx);
      struct timeval time;
      gettimeofday(&time,NULL);
      srand((time.tv_sec * 1000) + (time.tv_usec / 1000));
      init = 1;
    }

  static direzione ld = SINISTRA;

  int i = posi.tuki_y;
  int j = posi.tuki_x;
  int prossimo_vertice = -1;

  vertice_corrente = trova_vertice(i,j);

  if(
     vertice_corrente == vertice_goal||
     vertice_goal == -1
     )
    {
      do{
        vertice_goal = (double)rand()/((double)RAND_MAX)*NODI_LAB_POT;
      }
      while(labx[grafo[vertice_goal].linea][grafo[vertice_goal].columna] == 'J');
    }

  vertice_goal = evita_casa_fantasmi(vertice_goal);
```

```
  do
    {
      percorso_fuga = agri_astar
        (vertice_corrente,
         vertice_goal,
         grafo,&distanza_esatta,&euri,NODI_LAB_POT);
      if(!percorso_fuga)
        vertice_goal = (double)rand()/((double)RAND_MAX)*NODI_LAB_POT;
    }while(!percorso_fuga);

  prossimo_vertice = *percorso_fuga;
  free(percorso_fuga);

  if(grafo[vertice_corrente].ianua[SINISTRA] == prossimo_vertice)
    {
      ld = SINISTRA;
    }
  if(grafo[vertice_corrente].ianua[DESTRA] == prossimo_vertice)
    {
      ld = DESTRA;
    }
  if(grafo[vertice_corrente].ianua[SU] == prossimo_vertice)
    {
      ld = SU;
    }
  if(grafo[vertice_corrente].ianua[GIU] == prossimo_vertice)
        {
          ld = GIU;
        }
  return ld;
}
```

Listato gioca_tuki_respiro.c

```
/*_____________________________________________________________
 *
 * FILE: gioca_tuki_respiro.c
 */
#include "tuki5_modello.h"
#include "libagri.h"
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>

#define SCONOSCIUTO -2
#define GUINZAGLIO 10

#define NODI_LAB_POT 600

posizioni g_posi;
int n_vertici;
oggetto **lab;
static agri_Vertex grafo[NODI_LAB_POT];
agri_Vertex agri_Vertices_Colligati[NODI_LAB_POT];

int trova_vertice(int riga, int colonna)
{
  for(int i = 0; i<NODI_LAB_POT;i++)
    {
      if(grafo[i].linea == riga && grafo[i].columna == colonna)
        return grafo[i].index;
    }
  return -1;
}

int visitatus(int vertice)
{
  int r = grafo[vertice].linea;
  int c = grafo[vertice].columna;

  if(lab[r][c] == U ||lab[r][c] == V)
    return 0;
  return 1;
}
```

```
int phantasmatis_presentia(int vertice)
{
  int r = grafo[vertice].linea;
  int c = grafo[vertice].columna;

  int x_g[4];
  int y_g[4];
  x_g[0] = g_posi.blinky_x;
  x_g[1] = g_posi.pinky_x;
  x_g[2] = g_posi.inky_x;
  x_g[3] = g_posi.clyde_x;

  y_g[0] = g_posi.blinky_y;
  y_g[1] = g_posi.pinky_y;
  y_g[2] = g_posi.inky_y;
  y_g[3] = g_posi.clyde_y;

  for(int i = 0; i<4; i++)
    {
      int v = trova_vertice(y_g[i], x_g[i]);
      if(grafo[v].ianua[SINISTRA] == vertice ||
         grafo[v].ianua[DESTRA] == vertice ||
         grafo[v].ianua[SU] == vertice ||
         grafo[v].ianua[GIU] == vertice||
         grafo[v].index == vertice
         )
        {
          return 0;
        }
    }
  return 1;
}

int evita_casa_fantasmi(int vertice)
{
  int r = grafo[vertice].linea;
  int c = grafo[vertice].columna;
  if(r>=15 && r<=17 && c>=11 && c<=16)
    return 0;

  return vertice;
}

bool oggetto_accessibile(oggetto s)
{
  if(s == 'J' || s == 'U' || s == 'V')
    return true;
  else
    return false;
}

bool area_accessibile(int r, int c)
{
  if (r<3) return false;
  if(r>32) return false;
  if( r>=12 && r<=14 && (c<=4 || c>= 23)) return false;
  if( r>=18 && r<=20 && (c<=4 || c>= 23)) return false;
  return true;
}

double euri(int start, int goal)
{
  if(start<0)
    {
      exit(1);
    }

  int x = g_posi.tuki_x;
  int y = g_posi.tuki_y;
  int x_g[4];
  int y_g[4];
  x_g[0] = g_posi.blinky_x;
  x_g[1] = g_posi.pinky_x;
  x_g[2] = g_posi.inky_x;
  x_g[3] = g_posi.clyde_x;

  y_g[0] = g_posi.blinky_y;
  y_g[1] = g_posi.pinky_y;
  y_g[2] = g_posi.inky_y;
  y_g[3] = g_posi.clyde_y;

  int peso_g[]={700,700,700,700};
```

```
  int x1,x2,y1,y2;
  double d;
  x1 = agri_Vertices_Colligati[start].columna;
  y1 = agri_Vertices_Colligati[start].linea;
  x2 = agri_Vertices_Colligati[goal].columna;
  y2 = agri_Vertices_Colligati[goal].linea;

  d = (x1-x2)*(x1-x2)+(y1-y2)*(y1-y2);

  for(int i = 0; i<4; i++)
    {
      int v = trova_vertice(y_g[i], x_g[i]);
      if(grafo[v].ianua[SINISTRA] == start ||
         grafo[v].ianua[DESTRA] == start ||
         grafo[v].ianua[SU] == start ||
         grafo[v].ianua[GIU] == start||
         grafo[v].index == start
         )
        {
          return peso_g[i];
        }
    }
  return(sqrt(d));
}

void collega_tuki_nodi(oggetto **labx)
{
  int i_aux,j_aux;
  int k = 0;
  /* Genera i vertici del grafo */
  for(int i = 3; i<ALTEZZA-3; i++)
    {
      for(int j = 1; j<LARGHEZZA-1; j++)
        {
          if( oggetto_accessibile(labx[i][j]) && area_accessibile(i,j))
            {
              grafo[k].linea = i;
              grafo[k].columna = j;
              grafo[k].index = k;
              k++;
            }
        }
    }

  n_vertici = k;
  /* Collega i vertici del grafo */
  for(int k = 0; k<NODI_LAB_POT; k++)
    {
      int r = grafo[k].linea;
      int c = grafo[k].columna;
      if(c>0)
        grafo[k].ianua[SINISTRA] =  trova_vertice(r, c-1);
      if(c<LARGHEZZA - 1)
        grafo[k].ianua[DESTRA] = trova_vertice(r, c+1);
      if(r>0)
        grafo[k].ianua[SU] = trova_vertice(r-1, c);
      if(r<ALTEZZA-1)
        grafo[k].ianua[GIU] = trova_vertice(r+1, c);
    }
}

double distanza_esatta(int da_nodo, int a_nodo)
{
  int s = da_nodo;
  int g = a_nodo;
  static int distanze[NODI_LAB_POT][NODI_LAB_POT];
  static int init = 0;
  if(!init)
    {
      for(int i=0; i<NODI_LAB_POT;i++)
        for(int j=0; j<NODI_LAB_POT;j++)
          {
            distanze[i][j]=INFINITO;

            if(grafo[da_nodo].ianua[SINISTRA] == a_nodo ||
               grafo[da_nodo].ianua[DESTRA] == a_nodo ||
               grafo[da_nodo].ianua[SU] == a_nodo ||
               grafo[da_nodo].ianua[GIU] == a_nodo)
              distanze[i][j] = 1;
          }
      init = 1;
    }
```

```
  return (double)distanze[s][g];
}

direzione gioca_tuki(posizioni posi, oggetto **labx)
{
  static int * percorso_fuga = 0;
  static int vertice_goal = -1;
  static int init = 0;
  /* condivisione delle posizioni per accedere da euri **/
  g_posi = posi;
  lab = labx;

  int vertice_corrente = -1;

  if(!init)
    {
      collega_tuki_nodi(labx);
      struct timeval time;
      gettimeofday(&time,NULL);

      srand((time.tv_sec * 1000) + (time.tv_usec / 1000));
      init = 1;
    }

  static direzione ld = SINISTRA;

  int i = posi.tuki_y;
  int j = posi.tuki_x;

  int prossimo_vertice = -1;

  vertice_corrente = trova_vertice(i,j);

  /* STABILISCO IL PROSSIMO VERTICE */
  if(
     vertice_corrente == vertice_goal||
     vertice_goal == -1
     )
    {
      vertice_goal = agri_breadthfirstsearch
        (vertice_corrente,
         grafo,
         visitatus,
         NODI_LAB_POT
         );
    }
  percorso_fuga = agri_astar
    (vertice_corrente,
     vertice_goal,
     grafo,&distanza_esatta,&euri,NODI_LAB_POT);

  prossimo_vertice = *percorso_fuga;
  free(percorso_fuga);

  if(grafo[vertice_corrente].ianua[SINISTRA] == prossimo_vertice)
    {
      ld = SINISTRA;
    }
  if(grafo[vertice_corrente].ianua[DESTRA] == prossimo_vertice)
    {
      ld = DESTRA;
    }
  if(grafo[vertice_corrente].ianua[SU] == prossimo_vertice)
    {
      ld = SU;
    }
  if(grafo[vertice_corrente].ianua[GIU] == prossimo_vertice)
        {
          ld = GIU;
        }
  return ld;
}
```

Listato gioca_tuki_evita.c

```
/*_____________________________________________________________
 *
 * FILE: gioca_tuki_evita.c
```

```
 */
#include "tuki5_modello.h"
#include "libagri.h"
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>

#define SCONOSCIUTO -2
#define GUINZAGLIO 10
#define PROFONDITA_RICERCA 60
#define DISTANZA_SICUREZZA 16
#define TENTATIVI 50

#define NODI_LAB_POT 600

posizioni g_posi;
int n_vertici;
oggetto **lab;
static agri_Vertex grafo[NODI_LAB_POT];
agri_Vertex agri_Vertices_Colligati[NODI_LAB_POT];

int phantasmatis_presentia
(
 int start,
 agri_Vertex * agri_Vertices_Colligati,
 int (*fantasma_presente)(int ),
 int nmembri
 )
{
  static int iter;
  int precedente[nmembri];
  Ordo candidati = 0;
  int livello = 0;

  Ordo_insero_nodus(&candidati,start,1);

  while(candidati != 0)
    {
      livello++;
      int corrente = Ordo_pop(&candidati);

      /* restituisce il livello del nodo in cui ha trovato un fantasma */
      if(fantasma_presente(corrente)||livello>PROFONDITA_RICERCA)
        {
          return livello;
        }

      int vicino[PORTE];
      for(int i=0; i<PORTE; i++)
        {
          vicino[i] = agri_Vertices_Colligati[corrente].ianua[i];
        }

      for(int i =0; i<PORTE; i++)
        {
          int iv = vicino[i];

          if(iv == -1)continue;

          precedente[iv] = corrente;

          Ordo_insero_nodus(&candidati,iv,1);
        }
    }
  return 0;
}

int trova_vertice(int riga, int colonna)
{
  for(int i = 0; i<NODI_LAB_POT;i++)
    {
      if(grafo[i].linea == riga && grafo[i].columna == colonna)
        return grafo[i].index;
    }
  return -1;
}

int visitatus(int vertice)
{
```

```
  int r = grafo[vertice].linea;
  int c = grafo[vertice].columna;

  if(lab[r][c] == U ||lab[r][c] == V)
    return 0;
  return 1;
}

int phantasmatis(int vertice)
{

  int r = grafo[vertice].linea;
  int c = grafo[vertice].columna;

  if(r>=15 && r<=17 && c>=11 && c<=16)
    return 0;

  int x_g[4];
  int y_g[4];
  x_g[0] = g_posi.blinky_x;
  x_g[1] = g_posi.pinky_x;
  x_g[2] = g_posi.inky_x;
  x_g[3] = g_posi.clyde_x;

  y_g[0] = g_posi.blinky_y;
  y_g[1] = g_posi.pinky_y;
  y_g[2] = g_posi.inky_y;
  y_g[3] = g_posi.clyde_y;

  for(int i = 0; i<4; i++)
    {
      int v = trova_vertice(y_g[i], x_g[i]);
      if(grafo[v].ianua[SINISTRA] == vertice ||
         grafo[v].ianua[DESTRA] == vertice ||
         grafo[v].ianua[SU] == vertice ||
         grafo[v].ianua[GIU] == vertice||
         grafo[v].index == vertice
         )
        {
          return 1;
        }
    }
  return 0;
}

int evita_casa_fantasmi(int vertice)
{
  int r = grafo[vertice].linea;
  int c = grafo[vertice].columna;
  if(r>=15 && r<=17 && c>=11 && c<=16)
    return 0;

  return vertice;
}

bool oggetto_accessibile(oggetto s)
{
  if(s == 'J' || s == 'U' || s == 'V')
    return true;
  else
    return false;
}

bool area_accessibile(int r, int c)
{
  if (r<3) return false;
  if(r>32) return false;
  if( r>=12 && r<=14 && (c<=4 || c>= 23)) return false;
  if( r>=18 && r<=20 && (c<=4 || c>= 23)) return false;
  return true;
}

double euri(int start, int goal)
{
  if(start<0)
    {
      exit(1);
    }

  int x = g_posi.tuki_x;
  int y = g_posi.tuki_y;
  int x_g[4];
  int y_g[4];
```

```
  x_g[0] = g_posi.blinky_x;
  x_g[1] = g_posi.pinky_x;
  x_g[2] = g_posi.inky_x;
  x_g[3] = g_posi.clyde_x;

  y_g[0] = g_posi.blinky_y;
  y_g[1] = g_posi.pinky_y;
  y_g[2] = g_posi.inky_y;
  y_g[3] = g_posi.clyde_y;

  int peso_g[]={700,700,700,700};

  int x1,x2,y1,y2;
  double d;
  x1 = agri_Vertices_Colligati[start].columna;
  y1 = agri_Vertices_Colligati[start].linea;
  x2 = agri_Vertices_Colligati[goal].columna;
  y2 = agri_Vertices_Colligati[goal].linea;

  d = (x1-x2)*(x1-x2)+(y1-y2)*(y1-y2);

  for(int i = 0; i<4; i++)
    {
      int v = trova_vertice(y_g[i], x_g[i]);
      if(grafo[v].ianua[SINISTRA] == start ||
         grafo[v].ianua[DESTRA] == start ||
         grafo[v].ianua[SU] == start ||
         grafo[v].ianua[GIU] == start||
         grafo[v].index == start
         )
        {
          return peso_g[i];
        }
    }
  return(sqrt(d));
}

void collega_tuki_nodi(oggetto **labx)
{
  int i_aux,j_aux;
  int k = 0;
  /* Genera i vertici del grafo */
  for(int i = 3; i<ALTEZZA-3; i++)
    {
      for(int j = 1; j<LARGHEZZA-1; j++)
        {
          if( oggetto_accessibile(labx[i][j]) && area_accessibile(i,j))
            {
              grafo[k].linea = i;
              grafo[k].columna = j;
              grafo[k].index = k;
              k++;
            }
        }
    }

  n_vertici = k;
  /* Collega i vertici del grafo */
  for(int k = 0; k<NODI_LAB_POT; k++)
    {
      int r = grafo[k].linea;
      int c = grafo[k].columna;
      if(c>0)
        grafo[k].ianua[SINISTRA] =  trova_vertice(r, c-1);
      if(c<LARGHEZZA - 1)
        grafo[k].ianua[DESTRA] = trova_vertice(r, c+1);
      if(r>0)
        grafo[k].ianua[SU] = trova_vertice(r-1, c);
      if(r<ALTEZZA-1)
        grafo[k].ianua[GIU] = trova_vertice(r+1, c);
    }
}

double distanza_esatta(int da_nodo, int a_nodo)
{
  int s = da_nodo;
  int g = a_nodo;
  static int distanze[NODI_LAB_POT][NODI_LAB_POT];
  static int init = 0;
  if(!init)
    {
      for(int i=0; i<NODI_LAB_POT;i++)
        for(int j=0; j<NODI_LAB_POT;j++)
```

```
          {
            distanze[i][j]=INFINITO;

            if(grafo[da_nodo].ianua[SINISTRA] == a_nodo ||
               grafo[da_nodo].ianua[DESTRA] == a_nodo ||
               grafo[da_nodo].ianua[SU] == a_nodo ||
               grafo[da_nodo].ianua[GIU] == a_nodo)
              distanze[i][j] = 1;
          }
      init = 1;
    }
  return (double)distanze[s][g];
}


direzione gioca_tuki(posizioni posi, oggetto **labx)
{
  static int * percorso_fuga = 0;
  static int vertice_goal = -1;
  static int init = 0;
  g_posi = posi;
  lab = labx;

  int vertice_corrente = -1;

  if(!init)
    {
      collega_tuki_nodi(labx);
      struct timeval time;
      gettimeofday(&time,NULL);

      srand((time.tv_sec * 1000) + (time.tv_usec / 1000));
      init = 1;
    }

  static direzione ld = SINISTRA;

  int i = posi.tuki_y;
  int j = posi.tuki_x;

  int prossimo_vertice = -1;

  vertice_corrente = trova_vertice(i,j);
  char sicuro = 0;

  /* STABILISCO IL PROSSIMO VERTICE */
  if(
     vertice_corrente == vertice_goal||
     vertice_goal == -1
     )
    {
      /* cerca la pillola */
      vertice_goal = agri_breadthfirstsearch
        (vertice_corrente,
         grafo,
         visitatus,
         NODI_LAB_POT
         );
      /* verifica la presenza di fantasmi */

      sicuro = phantasmatis_presentia (vertice_goal,
                                       grafo,
                                       phantasmatis,
                                       NODI_LAB_POT
                                       );
      int tent = 0;
      while(sicuro<DISTANZA_SICUREZZA)
        {
          do{
            tent++;
            vertice_goal = (double)rand()/((double)RAND_MAX)*NODI_LAB_POT;
          }
          while(
                labx[grafo[vertice_goal].linea]
                [grafo[vertice_goal].columna] == 'J' &&
                tent<TENTATIVI
                );
          if(tent == TENTATIVI)
            vertice_goal = 1;

          vertice_goal = evita_casa_fantasmi(vertice_goal);
          sicuro = phantasmatis_presentia
            (vertice_goal,
```

```
            grafo,
            phantasmatis,
            NODI_LAB_POT
            );
        }
    }

  percorso_fuga = agri_astar
    (vertice_corrente,
     vertice_goal,
     grafo,&distanza_esatta,&euri,NODI_LAB_POT);

  prossimo_vertice = *percorso_fuga;
  free(percorso_fuga);

  if(grafo[vertice_corrente].ianua[SINISTRA] == prossimo_vertice)
    {
      ld = SINISTRA;
    }
  if(grafo[vertice_corrente].ianua[DESTRA] == prossimo_vertice)
    {
      ld = DESTRA;
    }
  if(grafo[vertice_corrente].ianua[SU] == prossimo_vertice)
    {
      ld = SU;
    }
  if(grafo[vertice_corrente].ianua[GIU] == prossimo_vertice)
        {
          ld = GIU;
        }

  return ld;
}
```

APPENDICE: CODICE COMPLETO TOOL A*

```
#include "tuki5_modello.h"
#include "libagri.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>

#define PORTE 10
#define INFINITO 9999
#define NODI_LAB_POT 44


/*_____________________________________________________________
 *|
 *| I nodi del grafo del labirinto
 */
agri_Vertex grafo[NNODI];
int b_iter[]={13,12,15,14,17,14,15,12,15,14,20,21,18,15,14,20};
int p_iter[]={12,13,16,19,22,23,17,11,8,7,6,5,4,3,2,0};
int passo;


double euristica_h(int start, int goal)
{
  int x1,x2,y1,y2;
  double d;
  x1 = grafo[start].columna;
  y1 = grafo[start].linea;
  x2 = grafo[goal].columna;
  y2 = grafo[goal].linea;
  //Euristica uguale al quadrato della distanza euclidea
  d = (x1-x2)*(x1-x2)+(y1-y2)*(y1-y2);
  /*
  if(start == 15)
    {
      return 200;
    }
  */
  if(grafo[b_iter[passo]].ianua[SINISTRA] == start ||
     grafo[b_iter[passo]].ianua[DESTRA] == start ||
     grafo[b_iter[passo]].ianua[SU] == start ||
     grafo[b_iter[passo]].ianua[GIU] == start)
    {
      return 300;
    }

  if(grafo[p_iter[passo]].ianua[SINISTRA] == start ||
     grafo[p_iter[passo]].ianua[DESTRA] == start ||
     grafo[p_iter[passo]].ianua[SU] == start ||
     grafo[p_iter[passo]].ianua[GIU] == start)
    {
      return 225;
    }


  return sqrt(d);
}



void Ordo_stampa(Ordo c)
{
  while(c)
    {
      printf(" Nodus %d, fscore %lf\n",c->index,c->prio);
      c = c->post;
    }
}


double distanza_esatta(int da_nodo, int a_nodo)
{
  //come nei codici precedenti
```

```
}

void collega_tuki_nodi()
{
  //come nei codici precedenti
}

/*_________________________________
 *
 * MAIN 1
 */

int main(int argc, char * argv[])
{
  int s = atoi(argv[1]);
  int g = atoi(argv[2]);
  collega_tuki_nodi();
  agri_Via pc = agri_astar(s,g,grafo,&distanza_esatta,&euristica_h,34);
  while(*pc>=0)
    {
      printf("->%d\n",*pc);
      pc++;
    }
}

/*_________________________________
 *
 * MAIN 2
 */
 /*
int main(int argc, char * argv[])
{
  int s = atoi(argv[1]);
  int g = atoi(argv[2]);

  collega_tuki_nodi();
  while(s != g)
    {
    printf("%d->\t",s);
      agri_Via pc = agri_astar(s,g,grafo,&distanza_esatta,&euristica_h,NODI_LAB_POT);
      s = *pc;
      while(*pc>=0)
        {
          printf("%d\t",*pc);
          pc++;
        }
      printf("\n");
    }
}
 */
/*_________________________________
 *
 * MAIN 3
 */
/*
int main(int argc, char * argv[])
{
  int s = atoi(argv[1]);
  int g = atoi(argv[2]);

  collega_tuki_nodi();

  while(s != g)
    {

      printf("%d->\t",s);
      agri_Via pc = agri_astar(s,g,grafo,&distanza_esatta,&euristica_h,NODI_LAB_POT);
      s = *pc;

      while(*pc>=0)
        {
          printf("%d\t",*pc);
          pc++;
        }
      printf(":%d :%d\n",b_iter[passo],p_iter[passo]);
      passo += 1;
    }
}
*/
```

SOLUZIONI

Box domande n.1

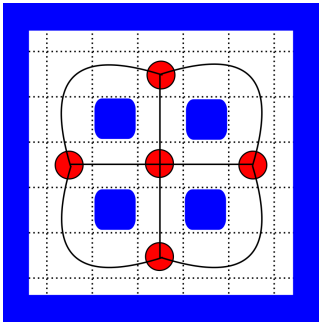
1. a
2. c
3. a

Box domande n.2

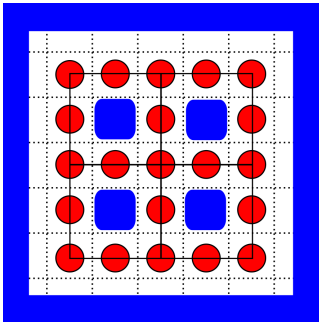
1. b
2. a
3. c

Box domande n.3

1. b
2. a



3. c



4. a

Box domande n.4

1. b
2. c
3. a

Box domande n.5

1. a
2. a
3. a

Box domande n.6

1. c
2. a
3. a
4. c

Box domande n.7

1. b
2. a
3. b

BIBLIOGRAFIA E RIFERIMENTI

- (Zachary) https://en.wikipedia.org/wiki/Zachary's_karate_club
-
- (p1) Pac-Man Conquers Academia: Two Decades of Research Using a Classic Arcade Game,IEEE TRANSACTIONS ON GAMES, VOL. 10, NO. 3, SEPTEMBER 2018
- Algorithm of ghost behavior in the game Pac-Man: https://weekly-geekly.github.io/articles/109406/index.html
- (wiliams) P. R. Williams, D. Perez-Liebana and S. M. Lucas, "Ms. Pac-Man Versus Ghost Team CIG 2016 competition," 2016 IEEE Conference on Computational Intelligence and Games (CIG), Santorini, 2016, pp. 1-8, doi: 10.1109/CIG.2016.7860446.
- (igraph) Gábor Csárdi, Tamás Nepusz: The igraph software package for complex network research. InterJournal Complex Systems, 1695, 2006.
- (Bähnemann) Rik Bähnemann, Nicholas Lawrance, Jen Jen Chung, Michael Pantic, Roland Siegwart, Juan Nieto: Revisiting Boustrophedon Coverage Path Planning as a Generalized Traveling Salesman Problem. arXiv:1907.09224
- (Choset) Choset H., Pignon P. (1998) Coverage Path Planning: The Boustrophedon Cellular Decomposition. In: Zelinsky A. (eds) Field and Service Robotics. Springer, London
- (Brooks) Rodney A.Brooks (1991): Intelligence without representation: MIT Artificial Intelligence Laboratory, 545 Technology Square, Rm. 836, Cambridge, MA 02139, USA

ALTRE PUBBLICAZIONI DI SCUOLA SISINI

Sfidare gli algoritmi

Introduzione alle reti neurali con esempi in linguaggio C

Reti neurali non supervisionate: il cognitrone di Fukushima