Un secondo problema : Quadrato Magico

- E' dato un quadrato di 10 caselle per 10 (in totale 100 caselle).
- Nello **stato iniziale** tutte le caselle sono vuote tranne la più in alto a sinistra che contiene il valore 1.
- **Problema**: assegnare a tutte le caselle un numero consecutivo, a partire da 1, fino a 100, secondo le seguenti regole:
  - A partire da una casella con valore assegnato x, si può assegnare il valore (x+1) solo ad una casella vuota che dista 2 caselle sia in verticale, che orizzontale, oppure 1 casella in diagonale.
- Applicare le strategie di ricerca cieche (su grafo e su albero) e quelle informate (greedy search, A*).

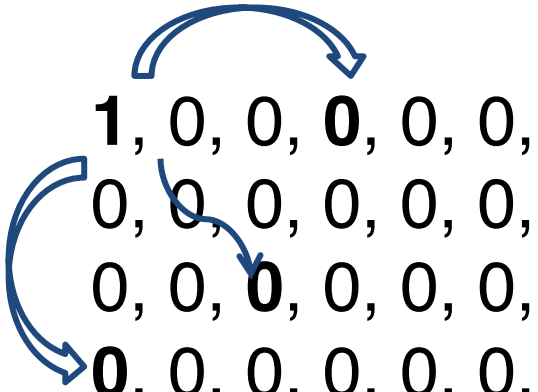
Un secondo problema : Quadrato Magico

- Come **funzione euristica**, si consideri il numero di celle non ancora assegnate
- Se il quadrato è ancora vuoto, per una casella generica ci sono 7 possibili caselle vuote su cui andare
- Perché le caselle sono 7 e non 8 (4 in diagonale e 4 in orizzontale/verticale)?
- Man mano che si riempie il quadrato, le caselle libere diminuiscono → diminuisce il fattore di ramificazione

- La profondità dell'albero è 100 (dobbiamo assegnare 100 numeri – 99 se il primo è già stato assegnato)

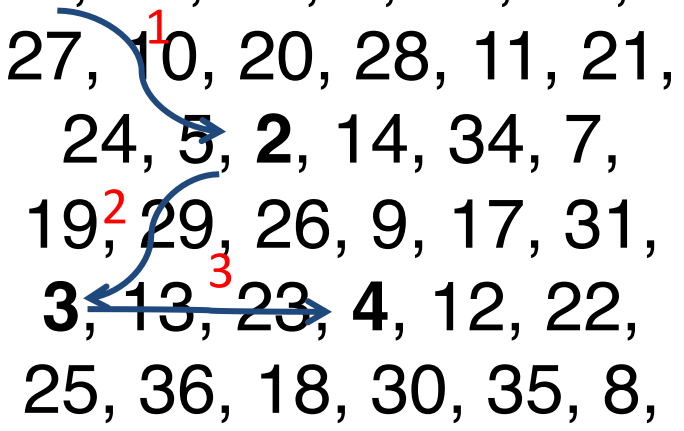
Un secondo problema : Quadrato Magico

Stato iniziale (6x6)



1, 0, 0, **0**, 0, 0,
0, 0, 0, 0, 0, 0,
0, 0, **0**, 0, 0, 0,
0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0,

Stato finale



1, 15, 33, 6, 16, 32,
27, 10, 20, 28, 11, 21,
24, 5, **2**, 14, 34, 7,
19, 29, 26, 9, 17, 31,
3, 13, 23, **4**, 12, 22,
25, 36, 18, 30, 35, 8,

Soluzione guidata

- Operatori: gli operatori devono permettere di spostarsi da una casella ad un'altra in direzione
  - Verticale: UP, DOWN
  - Orizzontale: LEFT, RIGHT
  - Diagonale: NE, SE, SW, NW

- Al più 8 operatori

- Test Obiettivo: **nessuna cella contiene il valore 0**
- Costo di ogni step = 1
- Funzione euristica: conta il n. di celle non assegnate nello stato corrente

Main

```
//creo un quadrato 10x10
SquareState initState = new SquareState(10);

//per strategie di ricerca cieche: uso il costruttore
public Problem(Object initialState,
               SuccessorFunction successorFunction,
               GoalTest goalTest)
Problem problem = new Problem(initState,
      new SquareSuccessorFunction(), initState);

//per strategie di ricerca informate: uso il
   costruttore public Problem(Object initialState,
   SuccessorFunction successorFunction, GoalTest
   goalTest, StepCostFunction stepCostFunction,
   HeuristicFunction heuristicFunction)
Problem problem = new Problem(initState,
        new SquareSuccessorFunction(),
        initState,initState,initState);
```

Main

```
Search search = … /*istanza della sottoclasse che
    implementa l'algoritmo di ricerca desiderato.
    Per le strategie informate:
    = new GreedyBestFirstSearch(new TreeSearch());
    = new AStarSearch(new TreeSearch()); */

SearchAgent agent = new SearchAgent(problem, search);


// semplici metodi di stampa dei risultati
printActions(agent.getActions());
printInstrumentation(agent.getInstrumentation());
```

Lo stato

```
public class SquareState implements
        GoalTest, StepCostFunction, HeuristicFunction {
//attributi: matrice rappresentante il quadrato,
    dimensione della matrice, ultima coordinata X e Y
    visitate, n. celle assegnate

//costruttore di default: inizializza gli attributi nello
    stato iniziale

//override isGoalState: controlla se lo stato finale è
    stato raggiunto

//override calculateStepCost: restituisce il costo di un
    passo

//override getHeuristicValue: restituisce la funzione
    euristica

//toString(): stampa la matrice quando il goal è raggiunto
    }
```

Funzione successore

```
public class SquareSuccessorFunction
             implements SuccessorFunction {

   public List getSuccessors(Object state) {

   // creare la lista dei successori (ArrayList)
   // recuperare lo stato corrente
   // testare se lo spostamento in un'altra cella
   (mossa) è possibile ed eseguire la mossa, generando
   uno stato successore come istanza di Successor
}
//metodo per verifica mossa consentita (spostamento
   ammissibile): per ognuna delle 8 direzioni di spostamento
   verificare che la cella di arrivo sia all'interno del
   quadrato e valga 0
//metodi per le mosse (8 possibili)
 per ognuna delle 8 direzioni di spostamento aggiornare
   l'ultima coordinata X e Y visitate dall'agente e il valore
   della cella della matrice
```

Un terzo problema : Il ponte degli U2 (esame 16 Dicembre 2005)

- Il complesso degli U2 sta per fare un concerto a Dublino.
- **Mancano 17 minuti** all'inizio del concerto ma, per raggiungere il palco, i membri del gruppo devono attraversare un piccolo ponte che è tutto al buio, disponendo di una sola torcia elettrica. **Sul ponte non possono andare più di due persone per volta.** La torcia è essenziale per l'attraversamento, per cui deve essere portata avanti e indietro (non può essere lanciata da una parte all'altra) per consentire a tutti di passare. **Tutti sono dalla stessa parte del ponte.**
- Ciascun componente del complesso cammina a una velocità diversa. I tempi individuali per attraversare il ponte sono:
  - Bono, 1 minuto
  - Edge, 2 minuti
  - Adam, 5 minuti
  - Larry, 10 minuti

Un terzo problema : Il ponte degli U2 (esame 16 Dicembre 2005)

- **Se attraversano in due, la coppia camminerà alla velocità del più lento.**
  - Ad esempio: se Bono e Larry attraversano per primi, quando arrivano dall'altra parte saranno trascorsi 10 minuti. Se Larry torna indietro con la torcia saranno passati altri dieci minuti, per cui la missione sarà fallita.

- Si stabilisca una funzione euristica ammissibile per questo problema e lo si risolva tramite l'algoritmo A* per i grafi, mostrando il grafo generato. Per limitare la dimensione dello spazio di ricerca**, si supponga che i componenti si muovano sempre in 2 in una direzione e sempre in 1 nella direzione opposta** (quando devono riportare indietro la torcia per prendere gli altri componenti della band).

Un terzo problema : Il ponte degli U2 (esame 16 Dicembre 2005)

- Per definire la funzione euristica *h'* si usi la seguente idea:
- Possono spostarsi solo 2 persone alla volta, per cui raggruppiamo le persone sulla riva di partenza (sinistra) in gruppi da 2. **Ordino i tempi di percorrenza dal più alto al più basso** e metto
  - nel primo gruppo i primi due
  - nel secondo gruppo il terzo ed il quarto.
- Ciascun gruppo si muoverà alla velocità del più lento, per cui per ogni gruppo prendo il massimo. **La funzione *h'* è data dalla somma dei tempi di percorrenza di ciascun gruppo. In altre parole, il tempo di percorrenza totale sarà almeno il tempo di percorrenza del più lento, più quello del terzo più lento** (il secondo più lento potrebbe essersi mosso con il più lento, per cui il suo tempo non viene calcolato).

Un terzo problema : Il ponte degli U2 (esame 16 Dicembre 2005)

- Nell'algoritmo A* per i grafi si considerano cammini alternativi per arrivare ad uno stesso stato; in caso di cammini alternativi viene tenuto il costo minore. Rappresentiamo con una freccia piena il cammino migliore e con una freccia tratteggiata i cammini peggiori.
- La funzione h' così definita è ammissibile? L'algoritmo troverà la strada migliore?

Soluzione guidata

- Operatori: gli operatori devono
  - portare indietro 4 persone *singolarmente*
  - portare avanti gruppi di 2 (per le combinazioni di persone a 2 a 2 tenere conto dell'ordinamento decrescente dei tempi individuali)

- Test Obiettivo: **Stato finale (1,1,1,1,1)** (persone+torcia)
- Costo di cammino: **Se attraversano in due, la coppia camminerà alla velocità del più lento.**
- Funzione euristica: funzione h'

Main

```
U2State initState = new U2State();
//istanza di Problem per strategia di ricerca informata
Problem problem = new Problem(
      initState,
      new U2SuccessorFunction(),
      initState,initState,initState);

Search search = new AStarSearch(new TreeSearch() o
                                new GraphSearch());

SearchAgent agent = new SearchAgent(problem, search);


// semplici metodi di stampa dei risultati
printActions(agent.getActions());
printInstrumentation(agent.getInstrumentation());
```

Lo stato

```
public class U2State implements
        GoalTest, StepCostFunction, HeuristicFunction {

//attributi: posizioni delle persone (4) e della torcia,
   tempi di spostamento individuali (4), tempo trascorso

//costruttore di default per lo stato iniziale e per lo
   stato generico a 6 arg.(posizioni e tempo trascorso)

//getter/setter per posizioni e tempo trascorso

//override isGoalState: controlla se lo stato finale è
   stato raggiunto (persone e torcia sul palco)

…
```

Lo stato

…

//override calculateStepCost: restituisce il costo di una traversata (vale per entrambe le direzioni), ovvero il tempo del più lento. Sfruttare la stringa "action" in ingresso al metodo per ricavare quale mossa è stata applicata (quale/quali persone ha/hanno attraversato).

//override getHeuristicValue: verifica chi si trova nel backstage, costruendo i gruppi da 2 (ci possono essere anche meno di 4 persone…). Si avvale di calculateStepCost per il calcolo della somma dei tempi dei 2 gruppi.

//verifica stato consentito: limite di tempo

Funzione successore

```
public class U2SuccessorFunction
             implements SuccessorFunction {

   public List getSuccessors(Object stato) {

   // creare lista dei successori (ArrayList)
   // recuperare lo stato corrente
   // distinguere i 2 casi dipendenti dalla posizione
   della torcia (in avanti si muovono in 2, indietro 1)
      //eseguire la mossa (spostamento di 1 o 2 persone
   dall'altra parte del ponte)
      //controllare stato consentito e aggiungere il
   successore
   }
   …
```

Funzione successore

```
…
//metodo per le mosse
// private U2State move(String actionString,
U2State current){}
```

Si possono codificare in **actionString le mosse** come concatenazione della direzione dello spostamento e delle persone coinvolte (1/2) tramite 3 caratteri.
Tenere conto del raggruppamento legato ai tempi di percorrenza per lo spostamento in avanti.
In base al valore della stringa, il metodo **move** aggiorna lo stato (posizioni e tempo trascorso).

```
}
```