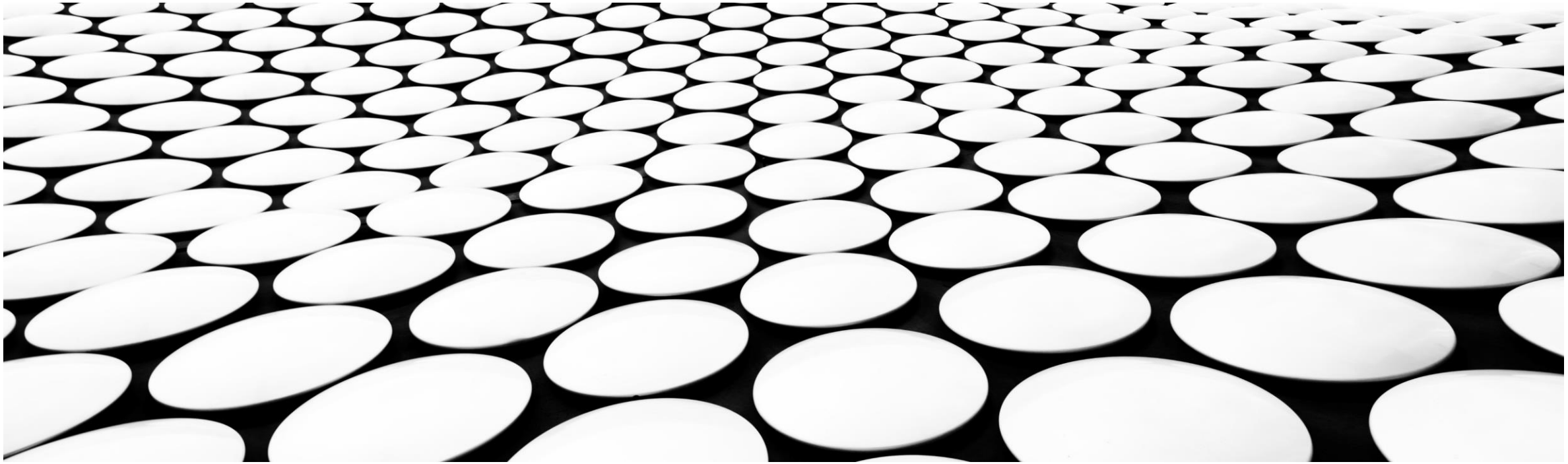
RISOLUZIONE PROBLEMI DI RICERCA

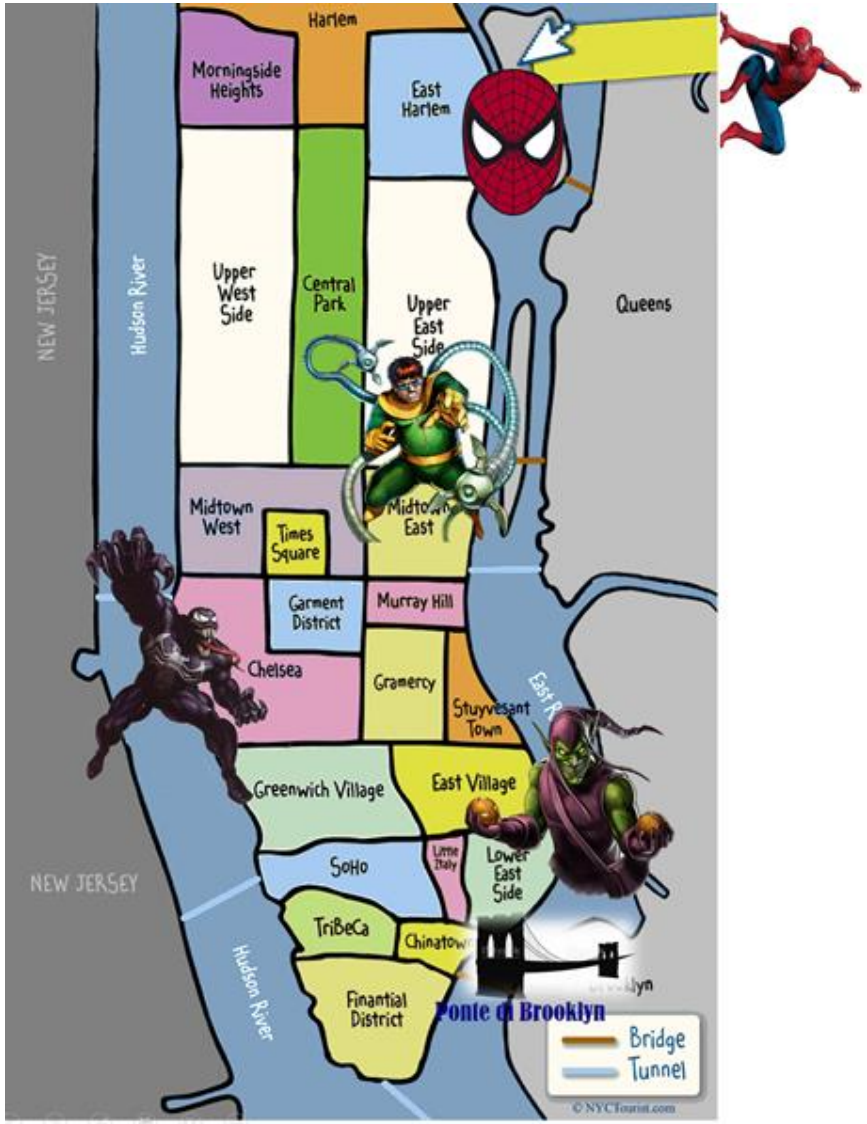
GUIDA ALLO SCRITTO D'ESAME



DOMANDE TIPICHE (1)

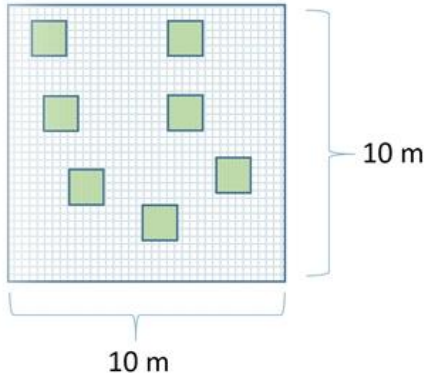
- **Domanda - Spiderman a Manhattan** Spiderman deve raggiungere il Ponte di Broklyn a New York, e non ha tempo da perdere: Mary Jane è in pericolo. La strada che lo divide dalla sua amata però è l'intera Manhattan visto che lui si trova nell'East Harlem, nella zona Nord della città. A complicare le cose c'è la presenza a New York di tre pericolosi criminali: Octopus, Venom e Goblin. Poichè Octopus si trova nel quartiere di Upper East Side, Venom a Chelsea e Goblin proprio nel Lower East Side, vicino al Ponte di Broklyn, lui deve liberarsi velocemente di tali ostacoli lungo la strada da Harlem al ponte. **Spiderman deve raggiungere nel modo più breve possibile i tre pericolsi criminali, librerarsi di loro poichè non gli facciano perdere tempo, e poi raggiungere May Jane.**

- Spiderman può muoversi tra i quartieri ma SOLO attraversando i loro confini, quindi saltando tra quartieri confinanti. Ogni quartiere ha una sua dimensione, quindi il passagio da uno all'altro fa perdere a Spiderman tempi diversi, proporzionali alla dimensione dei quartieri usati come partenza di un attraversamento. Nei molti modi di raggiungere il Ponte di Brooklyn, poichè consideriamo fermi i suoi antagonisti durate la suatraversata, Spiderman deve entrare nel quartiere dove uno dei criminali risiede uno dopo l'altro, trovando la strada più breve possibile che gli consenta di battere tutti e tre i suoi avversari.
  Si consideri nullo il tempo con cui Spiderman batte il suo avversario.
  Si richiede di:
  1. Programmare un agente che agisca come Spiderman, descrivendo (in pseudocodice, FOL, o Prolog o in Python) l'ambiente dove si muove Spiderman, la nozione di stato del gioco, il goal ed le informazioni che durante il ciclo del comportamento di SpiderMan egli scambia con l'ambiente
  2. Discutere una o più proprietà dell'ambiente che sono necessarie per calcolare il piano migliore, cioè il cammino più breve. Ad esempio, come si verifica il raggiungimento del goal, o come stabilire quale quartiere di volta in volta è necessario raggiungere perchè occupato da un supercriminale o perchè lungo la strada che ad esso conduce. Discutere anche le azioni possibili e le loro conseguenze nell'ambiente.
  3. Descrivere il programma che l'agente Spiderman usa per pianificare le sue azioni a Manhattan, e discutere quindi
  l'algoritmo di ricerca da lui applicato.
  4. (Facoltativo) Discutere come il programma dovrebbe esssere modificato nel caso in cui uno (o più) tra i supercriminali possa muoversi tra quartieri diversi, durante lo spostamento di Spiderman nella città.



Come in figura, Marina, residente a Londra, assiste estasiata alla caduta di banconote da 50 pound sopra al suo balcone di 10x10 metri, dove sta innaffiando le sue piante. Si accorge che il denaro cade attratto dalle piante ma che il concime con cui le innaffia fa dissolvere le singole banconote per una reazione chimica indesiderata. La Smith ha a disposizione mosse nelle 4 direzioni di 1 metro e raggiunta la pianta su cui cade la banconota la afferra prima che essa tocchi la pianta e la mette in tasca. Marina riesce a vedere la banconota più vicina alla distanza di 22 metri di altezza e quindi può sempre raggiungere la pianta su cui cade la successiva ed afferrarla all'altezza dei 2 metri. Nel frattempo deve percorrere il suo balcone, superando l'ostacolo delle diverse piante presenti su di esso.

La sua azione è dunque dedicata a raggiungere e prendere una banconota per volta, evitando le piante, appena questa si affaccia da un altezza di 22 metri.
Per massimizzare il suo vantaggio in denaro deve pianificare di volta in volta i suoi spostamenti in modo da determinare la pianta su cui cadrà la successiva banconota, raggiungerla evitando le piante presenti nel balcone (distribuite as esempio come nella figura a destra), e afferrarla mentre cade verticalmente, prima che, finendo nel vaso, essa si dissolva nell'acqua innaffiata.



Per realizzare un agente che riproduce il comportamento di Marina si richiede di:
Descrivere (usando lo pseudocodice, il linguaggio Python o il Prolog) l'ambiente ove si muove l'agente, la nozione di stato ed il goal

1. Discutere le proprietà necessarie che si intende rappresentare come modello del mondo (ad esempio la mappa o la raggiungibilità tra posizioni).
2. Descrivere il programma che l'agente usa per pianificare le sue azioni nell'ambiente.
3. Definire una funzione per l'azione di afferrare una certa banconota dipendente dalla sua posizione e la posizione (inclusa l'altezza) della banconota.
4. **Facoltativo**. Discutere la estensione del programma necessario a gestire il caso in cui esistano **due banconote** che scendono alla volta da due punti di caduta in generale diversi.

SCHEMA RISOLUTIVO

- Determinare la **natura del problema** e dell'**ambiente** (modello **P**erf **E**nv **A**ct **S**ens)
- Determinare la natura della **osservabilità** dell'ambiente
- Definire una **rappresentazione dell'ambiente, dello stato** e corrispondentemente della **percezione dell'agente**
  - **Grafo** (osservabile pienamente o parzialmente)
  - **Percezioni**: nodo ed eventualmente archi e tutte le distanze tra coppie di nodi
- Definire il **ciclo di vita dell'ambiente**: come cambia stato, come fornisce percezioni e come recepisce azioni dall'agente
- Definire il **ciclo di vita dell'agente** (osserva, pianifica, reagisce)
- Se la funzione pianifica è complessa (non on-line) sviluppo della relazione tra percezione e la algoritmica di ricerca che restituisce un piano (ad es. A*)
  - Definire attentamente la nozione di euristica
- Se la ricerca avviene in modalità on-line, concentrarsi sulla funzione utilità

AGENTE E AMBIENTE

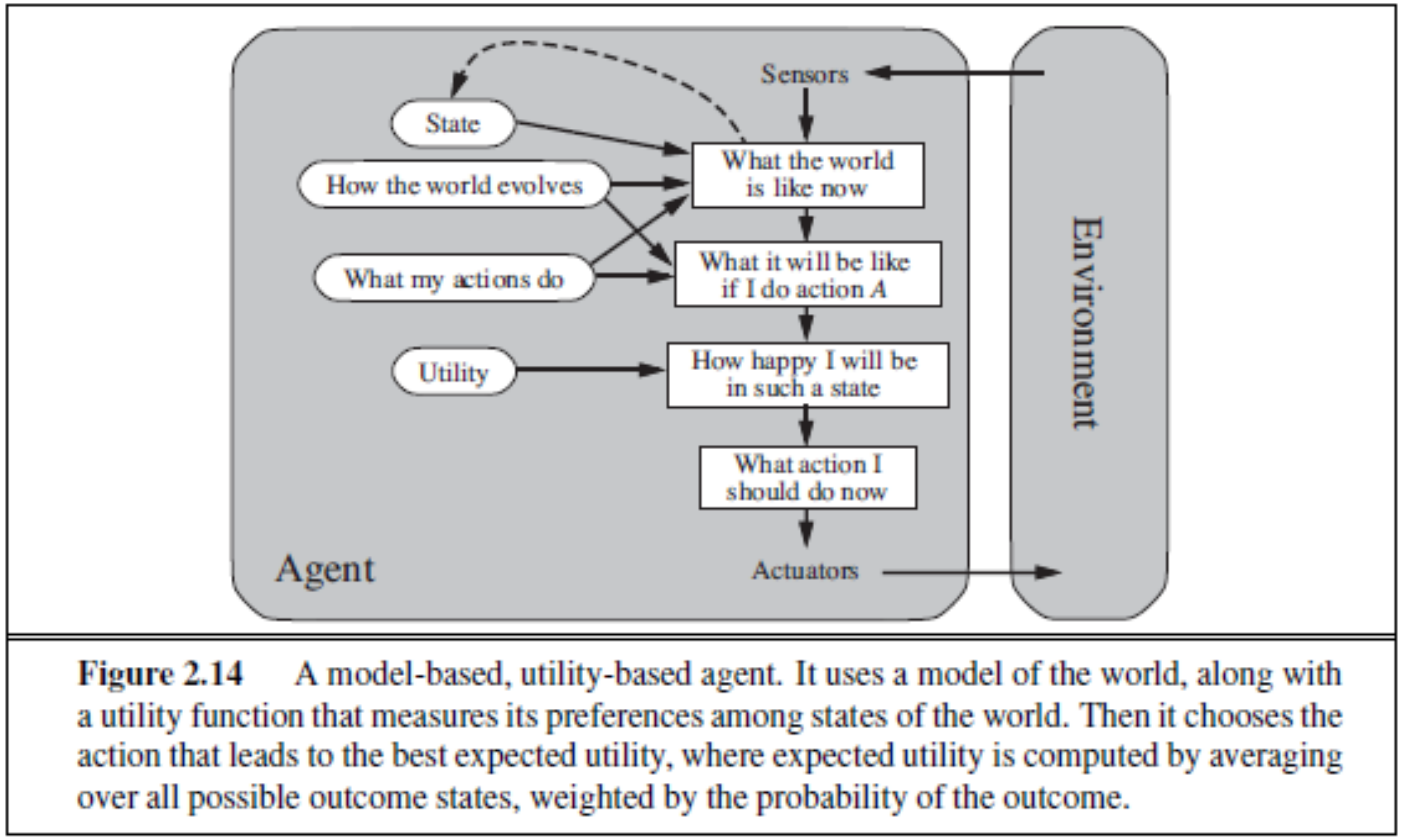


Figure 2.14 A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.

CICLO DI VITA DELL'AMBIENTE: SIMULATORE *MULTIAGENTE*

```
function RUN-EVAL-ENVIRONMENT(state,
                              UPDATE-FN, agents,
                                  PERFORMANCE-FN)
returns scores
local variables: scores  %(vector of size = #agents, all 0)
 repeat
    for each agent in agents do
       Percept[agent] ← GET-PERCEPT(agent, state)
    end
    for each agent in agents do
       Action[agent] ← PROGRAM[agent](Percept[agent])
    end
    state ← UPDATE-FN(Action, agents, state)
    scores ← PERFORMANCE-FN(scores, agents, state)
until TERMINATION(state)
return scores
```

CICLO DI VITA DELL'AMBIENTE: SIMULATORE *AGENTE SEMPLICE*

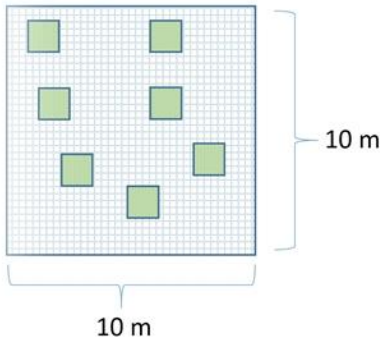
```
function RUN-EVAL-ENVIRONMENT(state,
                              UPDATE-FN, agent,
                                  PERFORMANCE-FN)
returns score
local variables: score  %(initially set to 0)
 repeat
    Percept ← GET-PERCEPT(agent, state)
    Action ← AGENTPROGRAM(Percept)
    state ← UPDATE-FN(Action, state)
    score ← PERFORMANCE-FN(score, state)
until TERMINATION(state)
return score
```

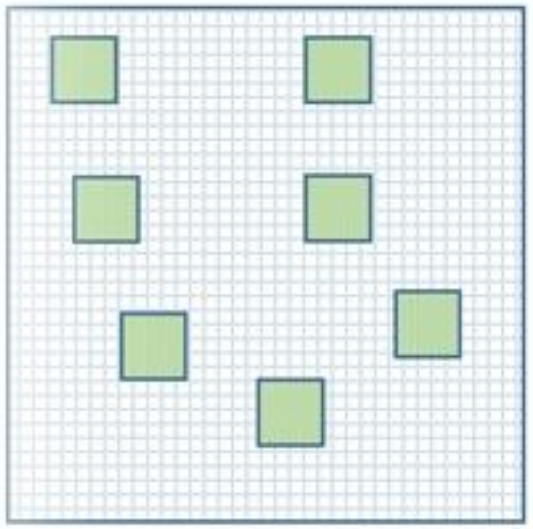
CICLO DI VITA DELL'AGENTE

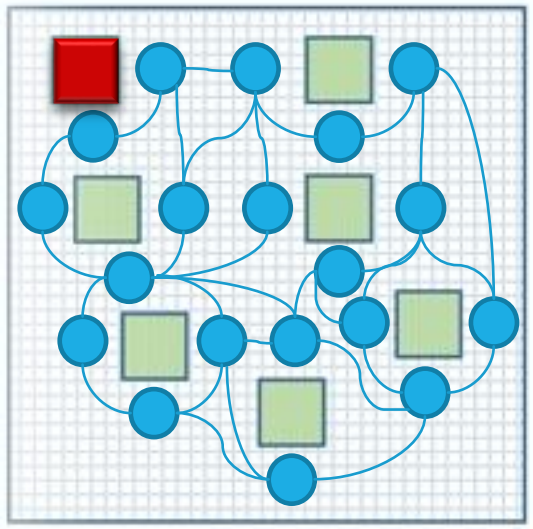
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

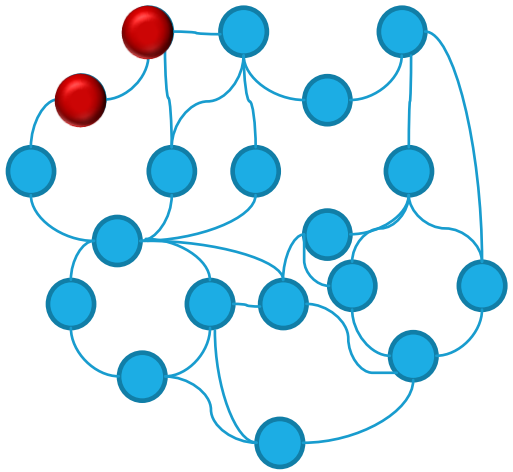
Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

ESEMPIO: MARTHA

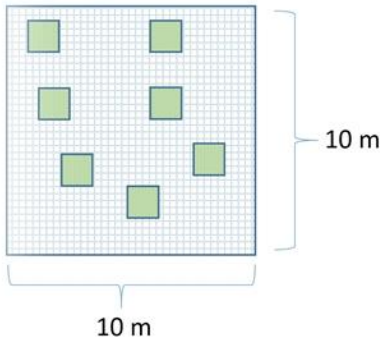


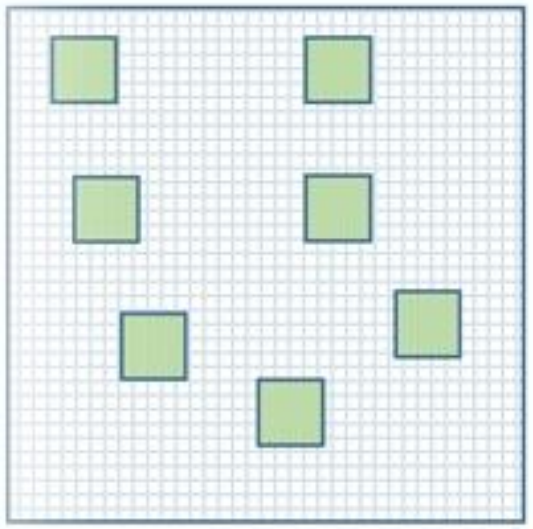


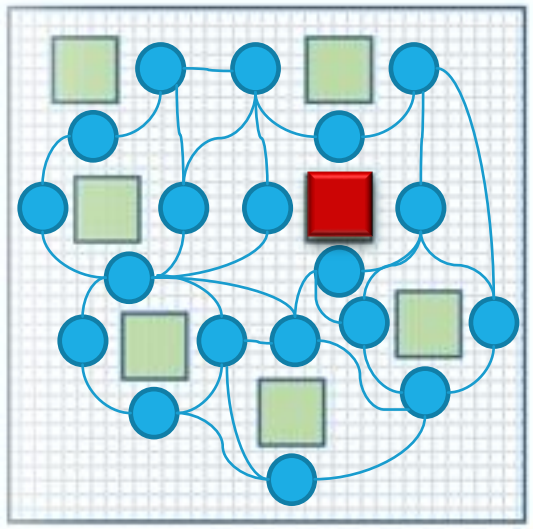


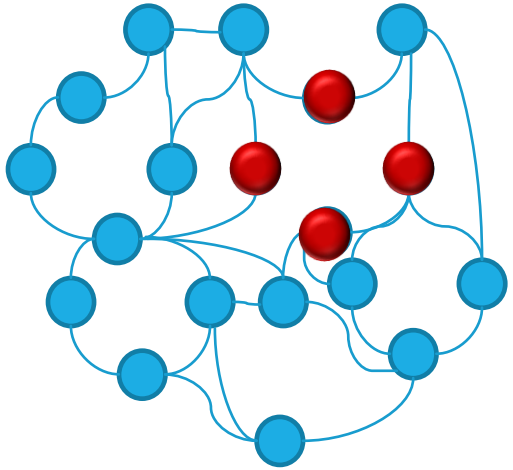


ESEMPIO: MARTHA

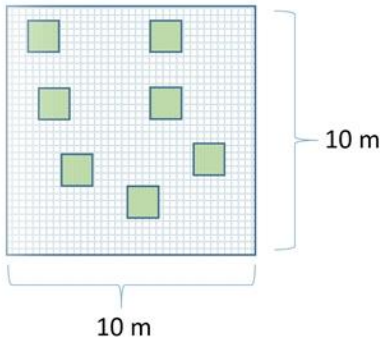


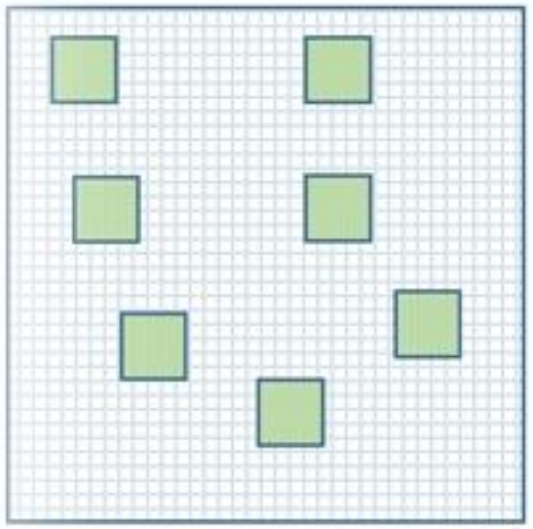


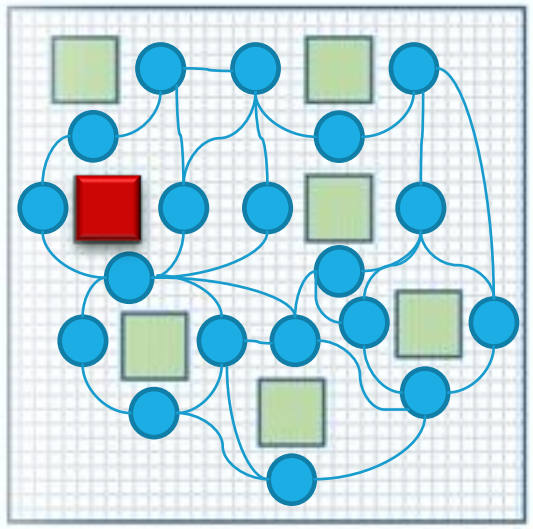


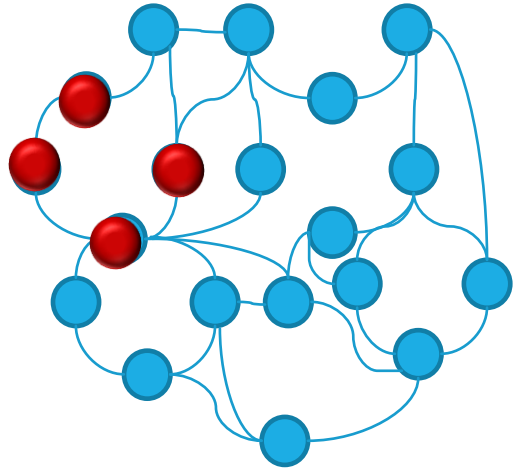


ESEMPIO: MARTHA

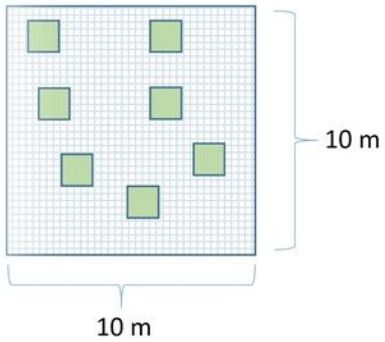




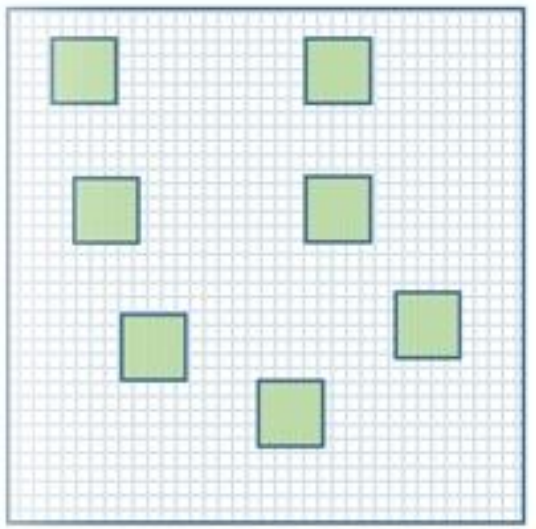


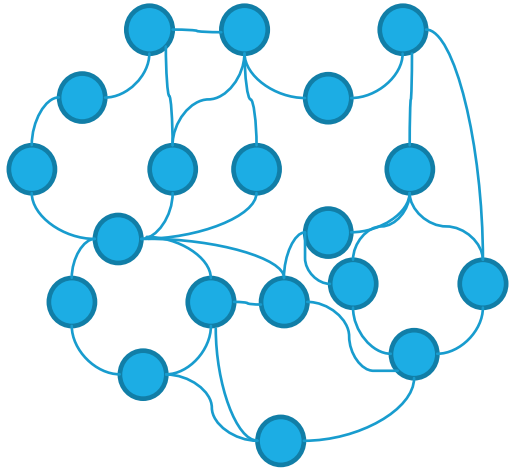


ESEMPIO: MARTHA

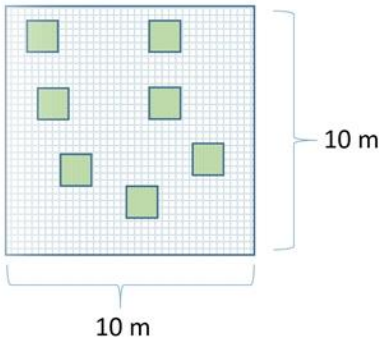


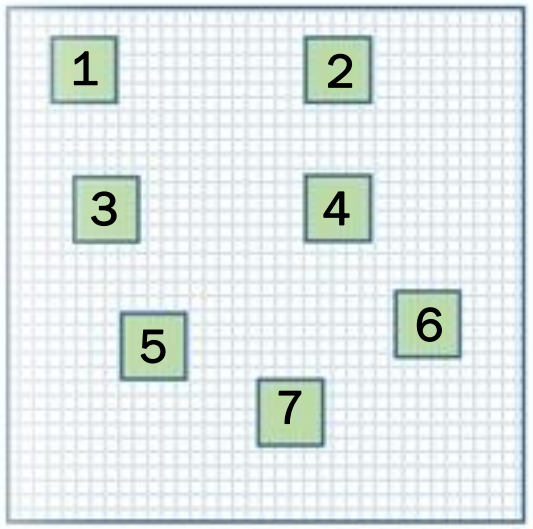
- Alcune semplificazioni introdotte, come esempi di ipotesi accettabili ai fine del test:
  - Non tutte le posizioni della matrice sono utili, quini i nodi sono funzionali al problema (ad es. sono solo attorno alle piante)
  - Non tutti gli archi sono utili, ed alcuni archi possono essere omessi
    - Tra nodi lontani
    - Tra nodi la cui connessione non sia utile ad alcuna soluzione
  - Il grafo può non essere generalizzabile ad altri problemi (ad esempio, spostamenti delle piante)

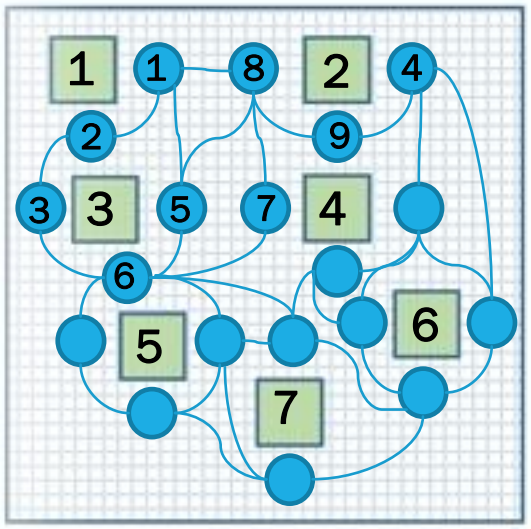




ESEMPIO: MARTHA

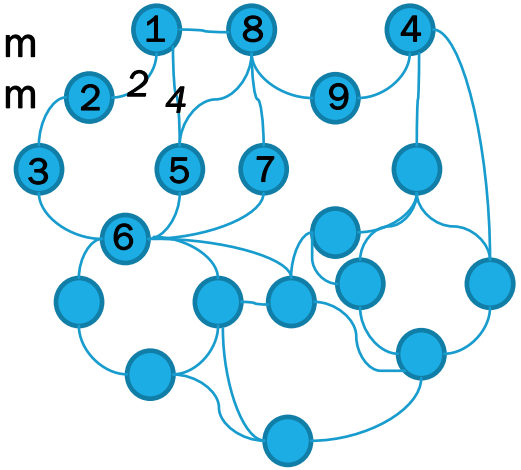






Modello archi

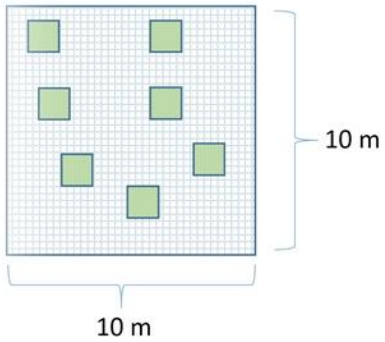
<1,2>: 2 m
<1,5>: 4 m
....
...



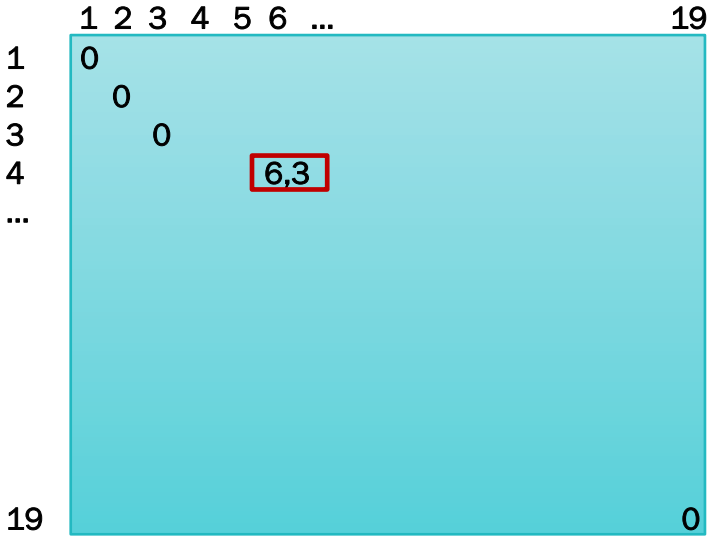
Modello delle Piante (Goal)

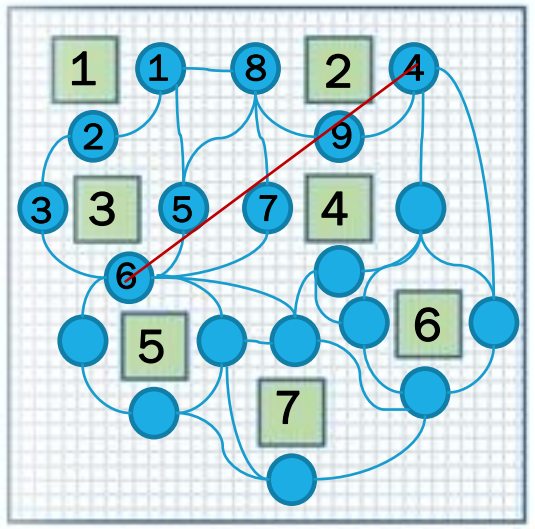
1: [1,2]
2: [4,8,9]
3: [2,3,5,6]
4: [7,9,...]
...

ESEMPIO: MARTHA



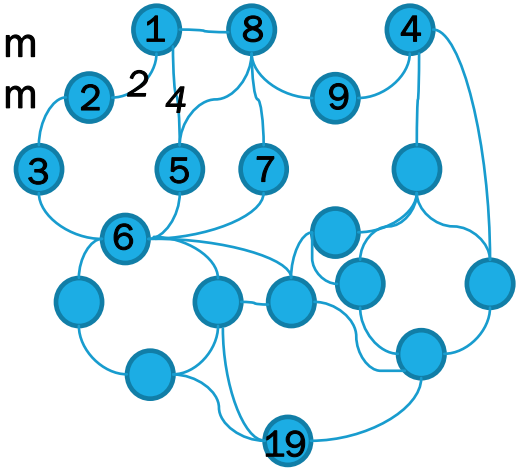
Euristica: Distanza in linea d'aria
Funzione euristica *h()*





Modello archi

<1,2>: 2 m
<1,5>: 4 m
....
...



Modello delle Piante (Goal)

1: [1,2]
2: [4,8,9]
3: [2,3,5,6]
4: [7,9,...]
...

RAPPRESENTAZIONE DELLO STATO

- Indice del nodo in cui è posizionato l'agente, NA
- Pianta Goal: P
- Indice dei nodi Goal [NG1, …,] al più 4
- Altezza della Banconota che cade sulla pianta (caso atomico): HB
- Score dell'AGENTE: come funzione delle Azioni effettuate e delle banconote raccolte
  - EN energia residua
  - DE denaro raccolto
- Stato Iniziale: [NA=?, P=?, [NG1(P),..,NG4(P)], HB=22, EN=1000, DE=0]
- Funzione agente: Stato, Grafo, Funzione Euristica

CICLO DI VITA DELL'AMBIENTE: SIMULATORE *AGENTE SEMPLICE*

```
function RUN-EVAL-ENVIRONMENT(state,
                              UPDATE-FN, agent,
                                  PERFORMANCE-FN)
returns score
local variables: score  %(initially set to 0)
 repeat
    Percept ← GET-PERCEPT(agent, state)
    Action ← AGENTPROGRAM(Percept)
    state ← UPDATE-FN(Action, state)
    score ← PERFORMANCE-FN(score, state)
until TERMINATION(state)
return score
```

CICLO DI VITA DELL'AMBIENTE: SIMULATORE *AGENTE SEMPLICE*

```
function RUN-EVAL-ENVIRONMENT(state,
                                  UPDATE-FN, agent,
                                             PERFORMANCE-FN)
returns (EN, DA)
local variables: (EN, DA)   %(initially set to (1000,0))
 repeat
    (NA,(NG1,..),HB)  ← GET-PERCEPT(agent, state)
    Action ← AGENTPROGRAM( NA,(NG1,..),HB)   %Nuovo Nodo o PrendiBanconota
    state ← UPDATE-FN(Action, state)
    (EN, DA) ← PERFORMANCE-FN(EN, DA, state)
until TERMINATION(state)
return score
```

CICLO DI VITA DELL'AGENTE MARTHA (GENERAL)

```
function MARTHA-AGENT(percept) returns an action
persistent:
        plan,    an action sequence, initially empty
        state, some description of the current world state
        goal,   a goal, initially null
        problem, a problem formulation, e.g. map of the fully observable environment

state ← UPDATE_STATE(state, percept )
if plan is empty then
    goal ← FORMULATE_GOAL(state)
    problem ← FORMULATE-PROBLEM(state, goal)
    plan ← SEARCH(problem)
    if plan = failure then return a null action
action ← FIRST(seq)
seq ← REST(seq)

return action
```

CICLO DI VITA DELL'AGENTE MARTHA (SPECIFIC)

```
function MARTHA-AGENT(NA,(NG1,..),HB) returns an action
persistent:
        plan,    an action sequence, initially empty
        state,   some description of the current world state
        goal,    a goal, initially null
        problem, a problem formulation, e.g. map of the fully observable environment

%%state ← UPDATE_STATE(state, NA,(NG1,..),HB)
if plan is empty then
    NG* ← FORMULATE_GOAL(NA,(NG1,..),HB) %Sceglie il nodo NG* che minimizza su i Dist(NA,Ngi)
    problem ← FORMULATE-PROBLEM(state, NG*)
    plan ← SEARCH(problem)  %A* con h() distanza in linea d'aria
    if plan = failure then return a null action   %Ad es. STOP
if NA=NG* then
    if HB>2 then action = STOP
   else action = CATCH
else
    action ← FIRST(seq)
    seq ← REST(seq)
return action
```

A*

- Definizione schema generale
- Scelta dell’euristica per il problema

RICERCA UC (RECALL)

f(n)=g(n)

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by f(n), with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher f(n) then
        replace that frontier node with child
```

Greedy Best First

Ricerca Best First

f(n)=h(n)

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE,
    frontier ← a priority queue ordered by f(n), with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
        if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                frontier ← INSERT(child, frontier)
            else if child.STATE is in frontier with higher f(n) then
                replace that frontier node with child
```

Greedy Best First

Ricerca A*

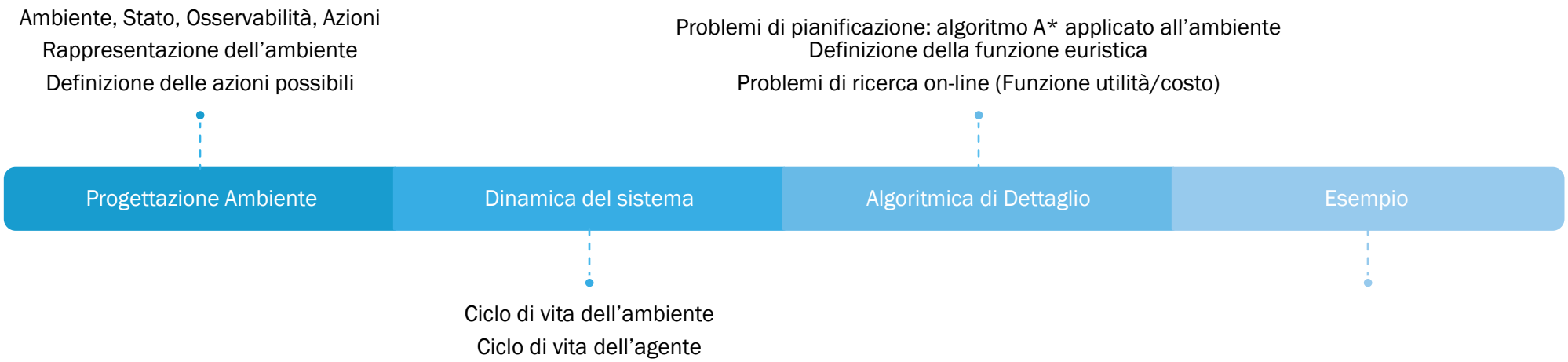
f(n)=g(n)+h(n)

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure

node ← a node with STATE = *problem*.INITIAL-STATE,
frontier ← a priority queue ordered by f(n), with *node* as the only element
explored ← an empty set
loop do
  if EMPTY?(*frontier*) **then return** failure
  node ← POP(*frontier*) /* chooses the lowest-cost node in *frontier* */
  if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
  add *node*.STATE to *explored*
  for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
    child ← CHILD-NODE(*problem*, *node*, *action*)
    if *child*.STATE is not in *explored* or *frontier* **then**
      frontier ← INSERT(*child*, *frontier*)
    else if *child*.STATE is in *frontier* with higher f(n) **then**
      replace that *frontier* node with *child*

A*

RECAP


Ambiente, Stato, Osservabilità, Azioni
Rappresentazione dell'ambiente
Definizione delle azioni possibili
Problemi di pianificazione: algoritmo A* applicato all'ambiente
Definizione della funzione euristica
Problemi di ricerca on-line (Funzione utilità/costo)
Progettazione Ambiente
Dinamica del sistema
Algoritmica di Dettaglio
Esempio
Ciclo di vita dell'ambiente
Ciclo di vita dell'agente