Problema 1

P: massimizza il denaro preso pianificando di volta in volta gli spostamenti in modo da determinare la pianta su cui cadrà la prossima banconota.

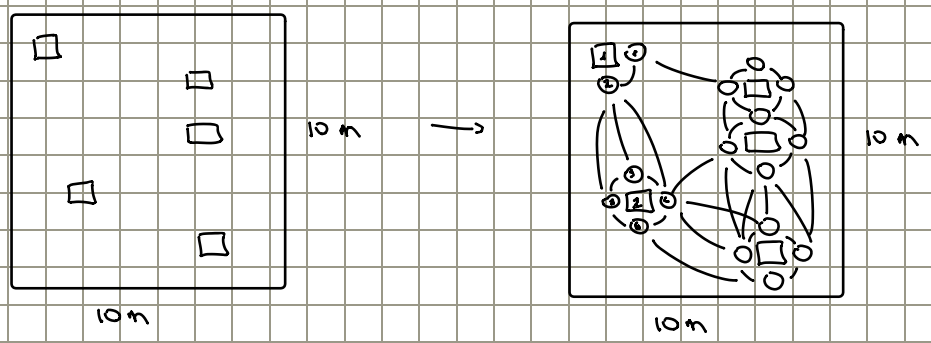
E: balcone 10x10 metri con all'interno piante che ostruiscono il passaggio e su cui cadono soldi.

A: gambe e braccia di Marta per andare e raccogliere i soldi.

S: occhi di Marta che vedono a 22 metri di distanza.

Modello ambiente

Possiamo vedere l'ambiente come una griglia 10 x 10 m dove ogni quadrato rappresenta una pianta



Possiamo rappresentare l'ambiente come un grafo dove i nodi sono le direzioni da dove possiamo raggiungere la pianta e gli archi sono il percorso che Marta può seguire per raggiungere una pianta. Importante notare come il grafo non è completamente connesso e che le posizioni accettabili sono quelle accanto alla pianta. L'ambiente è completamente osservabile.

archi: <1, 2> 2m
<2, 3> 5m
<2, 4> 7m
<4, 5> 2m
...

piante: 1: [1, 2]
2: [3, 4, 5, 6]
...

Descriviamo ora il ciclo di vita dell'ambiente dove il nostro agente è NA, il goal la pianta P i nodi vicino alla pianta P sono NG1 fino a NG4. EN rappresenta l'energia disponibile, DR invece il denaro raccolto. HB rappresenta l'altezza della banconota.

```
function RUN-EVAL-ENVIRONMENT(state, UPDATE-FN, NA, PERFORMANCE-FN) returns (EN, DR)
local variables: EN, DR   % inizialmente 1000, 0
repeat
  (NA, (NG1, ...), HB) <- GET-PERCEPT(NA, state)
   Action <- AGENT-MARTA((NA, (NG1, ...), HB))
```

```
  state ← UPDATE-FN (Action, state)
  (EN, DR) ← PERFORMANCE-FN ((EN, DR), state)
until TERMINATION (state)
return (EN, DA)
```

Ciclo di vita agente

```
function AGENT-MARTA (percept) returns action
persistent:
  plan, an action sequence, inizialmente vuota
  state, descrizione dello stato corrente del mondo
  goal, un goal
  problem, formulazione del problema,

if plan is empty then
  NG* ← FORMULATE-GOAL ((NA, (NG1, ...), AB))
  problem ← FORMULATE-PROBLEM ((NA, (NG1, ...), HB), NG*)
  plan ← SEARCH (problem)
  if plan = failure then return a null action
  if NA = NG* then
    if HB > 2 then action = STOP
    else action = CATCH
  else
    action ← FIRST (plan)
    plan ← REST (plan)
return action
```

Per la search utilizzeremo l'algoritmo A* con euristica di distanza euclidea

Definiamo ora l'algoritmo A*

```
function A*(problem) returns a solution or failure
  nodo ← nodo con stato = problem.INITIALSTATE
  frontiera ← coda di priorità ordinata in base a f(n) con all'inizio solo nodo "nodo"
  esplorati ← insieme dei nodi esplorati inizialmente vuoto
  loop do
    if frontiera is EMPTY? then return failure
    nodo ← POP(frontiera)
    if problem.GOALTEST(node.state) then return SOLUTION(node)
    add node.STATE to esplorati
    for each action in problem.ACTIONS(nodo.stato) do
      child ← CHILD-NODE(problem, node, action)
      if child.state non in frontiera or esplorati then
        frontiera ← INSERT(child.STATE)
      else if child.STATE is in frontiera con f(n) più alto allora
        replace that frontier node with child
```

In caso le banconote che scendono sono due il nostro obiettivo sarà raggiungere prima la banconota più vicina nel nodo più vicino all'altra banconota per poi una volta raccolta verificare se si c'è tempo per raggiungere anche la prossima e in caso andarci

Problema 2

Agente è il cavaliere

P: trovare il cammino minimo che permette al cavaliere di andare dalla principessa e tornare alla casella iniziale senza svegliare il drago.

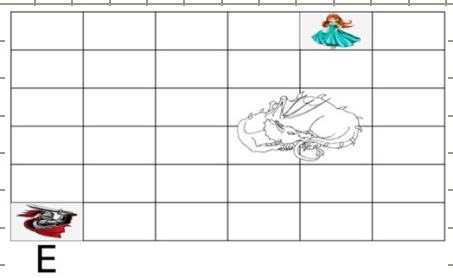
E: Una griglia 6x6 con il cavaliere posizionato nella cella (1,1), con il drago che occupa 4 caselle e la principessa posizionata in una casella.

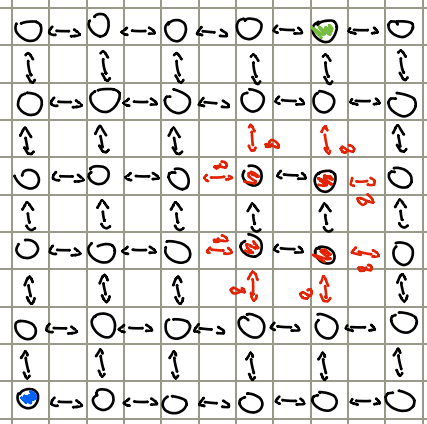
A: gambe del cavaliere.

S: occhi del cavaliere.

Modello dell'ambiente

L'ambiente è una griglia 6x6 che possiamo vedere come un grafo in cui ogni casella ha un arco collegato alla casella adiacente, tutti gli archi che collegano le celle del drago con le celle adiacenti avranno peso $+\infty$. Il nostro goal iniziale è quello di arrivare alla casella della principessa, senza passare sulle caselle dov'è presente il drago successivamente il nuovo goal sarà tornare alla posizione iniziale. Ogni cella rappresenta uno stato dell'ambiente





In verde è segnato il nostro primo goal, in blu la posizione iniziale e gli archi rossi sono gli archi il quale peso è infinito. Il nostro prossimo goal sarà tornare nella posizione iniziale evitando comunque il drago.

Da notare come il peso di tutti gli altri archi sarà 1 e lo score finale dovrà essere ridotto di 1. Le uniche azioni che il nostro agente potrà compiere sono 4

"N" "S" "E" "W" "N" si sposta da $(x,y) \to (x, y+1)$

"S" " " " $(x,y) \to (x, y-1)$

" E " " " " $(x,y) \rightarrow (x-1, y)$

" W " " " " $(x,y) \rightarrow (x+1, y)$

Soltanto se queste azioni sono disponibili.

Descriviamo ora il ciclo di vita dell'agente e dell'ambiente

```
function RUN-EVAL-ENVIR (state, UPDATE-FN, CV, PERFORMANCE-FN) returns CE
local variables CE %inizialmente 0
repeat
 (AUPOS, DRPOS, PRPOS, EXPOS) ← GET-PERCEPT (CV, state)
    Action ← CV-PROG (AUPOS, DRPOS, PRPOS, EXPOS)
    state ← UPDATE-FN (state, Action)
    CE ← PERFORM-FN (state, CE)
until TERMINATION (state)
return CE

function CV-PROGRAM (percept) returns action
persistent:
    plan, sequenza di azioni          inizialmente vuota
    state, stato corrente del mondo
    goal, nodo goal
    problem, problema
if plan is EMPTY? then
   NG ← FORMULATE-GOAL((AUPOS, EXPOS, DUPOS, PRPOS))
   problem ← FORMULATE-PROBLEM((AUPOS, EXPOS, DUPOS, PRPOS), NG)
   plan ← SEARCH(problem)     % A*(problem, f(n) = g(n) + h(n)) h euristica di dist. manhattan
if plan = failure then return null action
else
   if NG = PRPOS then catch
   else if NG = EXPOS  then exit
   else
```

```
    action ← FIRST(plan)
    plan ← REST(plan)
  return action

function A* (problem, f(n)) returns action or failure
nodo ← nodo con stato problem.INITIALSTATE
frontiera ← coda con priorità con inizialmente solo "nodo"
esplorati ← Insieme inizialmente vuoto
loop do
  if frontiera is EMPTY? return failure
  nodo ← POP(frontiera)
  if problem.GOALTEST(nodo.STATE) then return SOLUTION(nodo)
  add nodo to esplorati
  for each action in problem.ACTIONS(nodo.stato) do
    child ← CHILD-NODE(problem, nodo, action)
    if child.state non in frontiera o esplorati then
      frontiera ← INSERT(child, frontiera)
    else if child.state in frontiera con f(n) più grande allora
      replace that node with child.
```

```
function RUN-EVAL-ENV(state, UPDATE, agent, PERF) returns score
local variables: score ← initially 0
repeat
  Percept ← GET-PERCEPT(state, agent)
  Action ← AGENT-PROGRAM(percept)
  state ← UPDATE(state, Action)
  score ← PERF(state, score)
until TERMINATED(state)
return Action

function AGENT-PROGRAM(percept) returns action
  persistent
    state    state we can
    goal     initial 0
    problem  formulated prob
    plan     stay empty initial 0
  if plan is empty then
    goal ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROB(state, goal)
    plan ← SEARCH(problem)
    if plan = failure then return null action
  action ← FIRST(plan)
  plan ← REST(plan)
  return action
```

```
function A*(problem, f(n))
  node ← NODE(problem.INITIAL-STATE)
  frontier ← priority queue ordered by f, with node
  explored ← empty set of states
  loop do
    if frontier is empty then return failure
    node ← pop(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 is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child is in frontier with higher f then
        replace that node with frontier node child
```

Problema 3

L'agente è SPIDERMAN

P: Far arrivare l'agente sul ponte passando nei nodi dove sono presenti i cattivi usando però la strada più breve possibile.

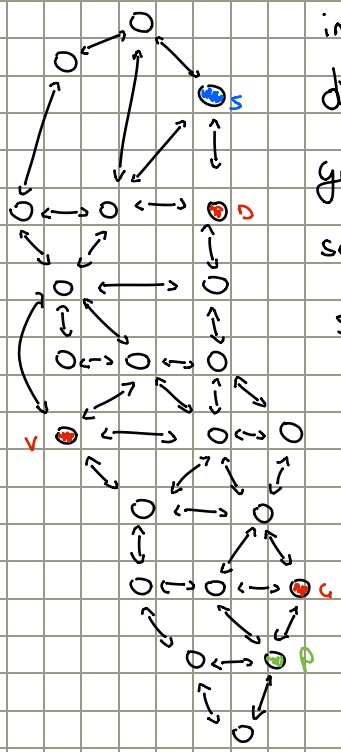
E: Grafo dove i nodi sono i vari quartieri e gli archi sono hanno come costo la transizione da quel quartiere a quello confinante

A: braccia e gambe

S: occhi

Modellazione dell'ambiente

Abbiamo quindi un grafo corrispondente alla mappa di New York dove ogni quartiere rappresenta un nodo. Il peso degli archi avranno costo variabile in base al tempo che il nostro agente impiega per passare il confine. Ci sono diversi goal: il primo è quello di raggiungere DR OCTOPUS nel nodo DPOS per sconfiggerlo, il successivo goal è raggiungere VENOM nel nodo VPOS e successivamente il goal sarà sconfiggere GREEN GOBLIN nel nodo GPOS solo allora il nostro ultimo goal sarà raggiungere il ponte in posizione PPOS. E' importante notare che il grafo non è completamente connesso e che poiché i cattivi sono tutti fermi e già sappiamo la loro posizione allora l'ambiente è deterministico e completamente osservabile.



Lo stato iniziale dell'agente sarà East Harlem con tutti i nemici non sconfitti.

Lo stato finale sarà China Town con tutti i nemici sconfitti.

Le azioni che SPIDERMAN potrà compiere sono o di spostamento o di combattimento.

Stato iniziale: (EH, (DPOS, VPOS, GPOS, PPOS))

Stato finale: (PPOS, (DPOS, VPOS, GPOS, PPOS))

Score: DIST

NG = nodo goal

Ciclo di vita dell'ambiente

```
function RUN-EVAL-ENVIR (state, UPDATE-FN, agent, PERFORMANCE-FN) returns DIST
local variable DIST   % inizialmente a 0
repeat
   (EA, (D,V,G,P)) ← GET-PERCEPT(EA, state)
    Actions ← SP-PROGRAM(EA, (D,V,G,P))
   (EA, (D,V,G,P)) ← UPDATE-FN((EA, (D,V,G,P)), Action)
    DIST ← PERFORMANCE-FN((EA, (D,V,G,P)), DIST)
until termination((EA, (D,V,G,P)))
return DIST
```

Ciclo di vita dell'agente

```
function SP-PROGRAM((EA, (D,V,G,P)) returns action
persistent:
   state   % stato del mondo corrente
   goal   % indica il goal dove dobbiamo arrivare
   problem   % formulazione del problema
   plan   % seq di azioni inizialmente vuota
if plan is empty then
   NG ← FORMULATE-GOAL((EA, (D,V,G,P))
   problem ← FORMULATE- ((EA, (D,V,G,P)), NG)
   plan ← SEARCH(problem)   % A* euristica dist. manatthan
if plan = failure then return null action
else if NG = D o NG = V o NG = G then return action ATTACK
else if NG = P then return action SAVE
else
  action ← FIRST(plan)
  plan ← rest(plan)
return action
```