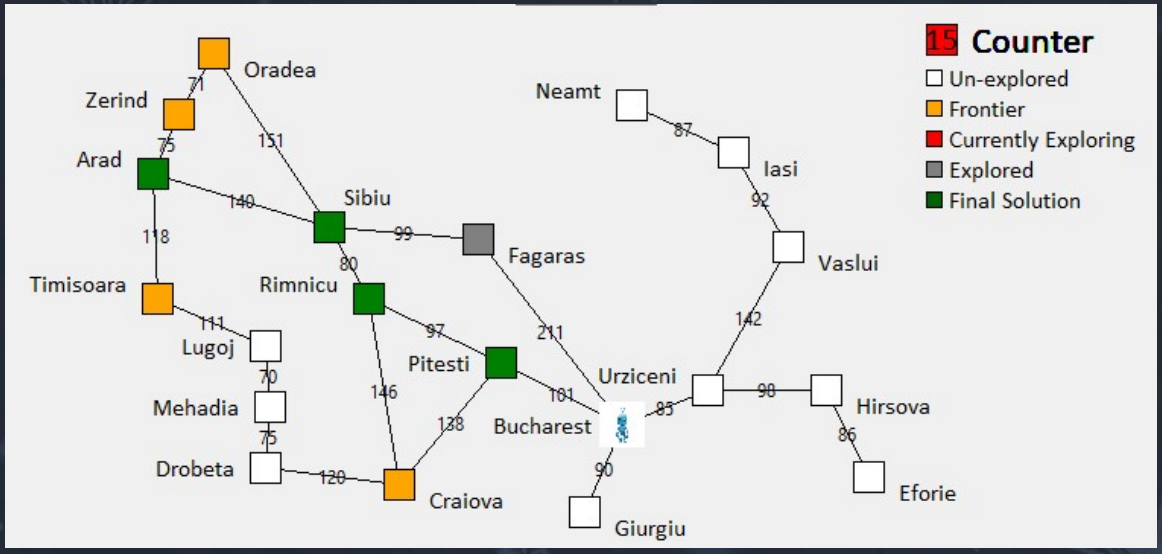
Search in Python and Prolog

Università degli Studi di Roma Tor Vergata

C.D. Hromei

Obiettivo

- Data la mappa della Romania, come già vista in altre esercitazioni, bisogna trovare il cammino minimo da una città A ad una città B **in PROLOG!**



Python

1. Creare la mappa con le città e gli archi pesati

```
def create_map(root):
    """This function draws out the required map."""
    global city_map, start, goal
    romania_locations = romania_map.locations
    width = 750 #ex 750 """
    height = 600 # ex 670 """
    margin = 10
    city_map = Canvas(root, width=width, height=height)
    update_counter()
    make_lines(city_map, romania_locations)
```

Python

1. Creare la mappa con le città e gli archi pesati
2. Definire l'algoritmo di ricerca A*

```
def astar_search(problem, h=None):
    """A* search is best-first graph search with f(n) = g(n)+h(n).
    You need to specify the h function when you call astar_search, or
    else in your Problem subclass."""
    h = memoize(h or problem.h, 'h')
    f = lambda n: n.path_cost + h(n)

    f = memoize(f, 'f')
    node = Node(problem.initial)
    frontier = PriorityQueue('min', f)
    frontier.append(node)
    explored = set()
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node
        explored.add(node.state)
        for child in node.expand(problem):
            if child.state not in explored and child not in frontier:
                frontier.append(child)
            elif child in frontier:
                if f(child) < frontier[child]:
                    del frontier[child]
                    frontier.append(child)
    return None
```

Python

1. Creare la mappa con le città e gli archi pesati
2. Definire l'algoritmo di ricerca A*
3. Applicare A* dalla città di partenza e spostarsi. Ogni volta, controllare se la città di arrivo corrisponde al goal. Se sì, restituire la soluzione

```
node = search_algorithm(romania_problem)
if node is not None:
    final_path = node.solution()
    final_path.insert(0, start.get())

    # compute total cost
    total = 0
    for i in range(len(final_path)-1):
        total += romania_map.get(final_path[i], final_path[i+1])
    print(f"Algorithm {algo.get()} needed {counter} counts!")
    print(f"Distance from {start.get()} to {goal.get()} is {total}.")
    print("_____")

    display_final(final_path)
    move_image(final_path)
    next_button.config(state="disabled")
```

Prolog

1. Necessario definire le città con le proprie posizioni e gli archi con i pesi, come fatti.

```
loc('Arad', 91, 492).
loc('Bucharest', 400, 327).
loc('Craiova', 253, 288).
loc('Drobeta', 165, 299).
loc('Eforie', 562, 293).
loc('Fagaras', 305, 449).
loc('Giurgiu', 375, 270).
loc('Hirsova', 534, 350).
loc('Iasi', 473, 506).
loc('Lugoj', 165, 379).
loc('Mehadia', 168, 339).
loc('Neamt', 406, 537).
loc('Oradea', 131, 571).
loc('Pitesti', 320, 368).
loc('Rimnicu', 233, 410).
loc('Sibiu', 207, 457).
loc('Timisoara', 94, 410).
loc('Urziceni', 456, 350).
loc('Vaslui', 509, 444).
loc('Zerind', 108, 531).
```

```
arc('Arad', 'Zerind', 75).
arc('Arad', 'Sibiu', 140).
arc('Arad', 'Timisoara', 118).
arc('Bucharest', 'Urziceni', 85).
arc('Bucharest', 'Pitesti', 101).
arc('Bucharest', 'Giurgiu', 90).
arc('Bucharest', 'Fagaras', 211).
arc('Craiova', 'Drobeta', 120).
arc('Craiova', 'Rimnicu', 146).
arc('Craiova', 'Pitesti', 138).
arc('Drobeta', 'Mehadia', 75).
arc('Eforie', 'Hirsova', 86).
arc('Fagaras', 'Sibiu', 99).
arc('Hirsova', 'Urziceni', 98).
arc('Iasi', 'Vaslui', 92).
arc('Iasi', 'Neamt', 87).
arc('Lugoj', 'Timisoara', 111).
arc('Lugoj', 'Mehadia', 70).
arc('Oradea', 'Zerind', 71).
arc('Oradea', 'Sibiu', 151).
arc('Pitesti', 'Rimnicu', 97).
arc('Rimnicu', 'Sibiu', 80).
arc('Urziceni', 'Vaslui', 142).
```

Prolog

2. Definire un predicato che calcoli la soluzione migliore e il costo

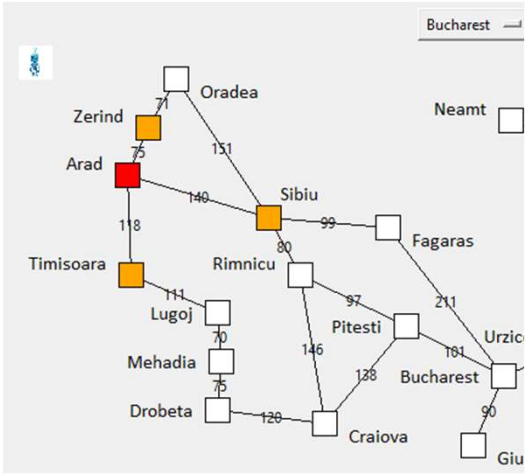
```
/* main predicate */
makepath(Source, Target, Sol) :-
    abolish(goal/1),
    assert(goal(Target)),
    solve(Source, Sol, Cost),
    nl,
    write('The Solution is: '), pretty_write(Sol), write(' with Cost: '), write(Cost),nl,
    nl, write('Done!!\n').
```

```
/* A-star Solving */
solve(Start, Soln, Function) :-
    f_function_new(Start, 0, 0, F), /* Heuristic function */
    /* Search over active successors/states
        One successor is a 4-ple:
        - Current Node
        - Current Depth D
        - Current Incurred Cost
        - Current Estimated Cost F
        - Current Solution (in reverse order) */
    search([(Start,0, 0, F,[])], S, Function),
    reverse(S,Soln).
```

```
/* f defined as arc cost plus heuristic cost: g(n) + h(n) */
f_function_new(State, IncurredCost, ArcCost, F) :-
    h_function_ed(State, IncurredCost, H),
    F is ArcCost + H.

/* heuristics of local arc cost */
h_function(_State, Cost, Cost).
/* heuristics: distance to target */
h_function_ed(State, _, H) :-
    euclidean_dist_to_goal(State, H).

euclidean_dist_to_goal(State, H) :-
    goal(Target),
    loc(State, X1,Y1),
    loc(Target, X2, Y2),
    H is sqrt( (X1-X2)*(X1-X2) + (Y1-Y2)*(Y1-Y2) ).
```

Prolog

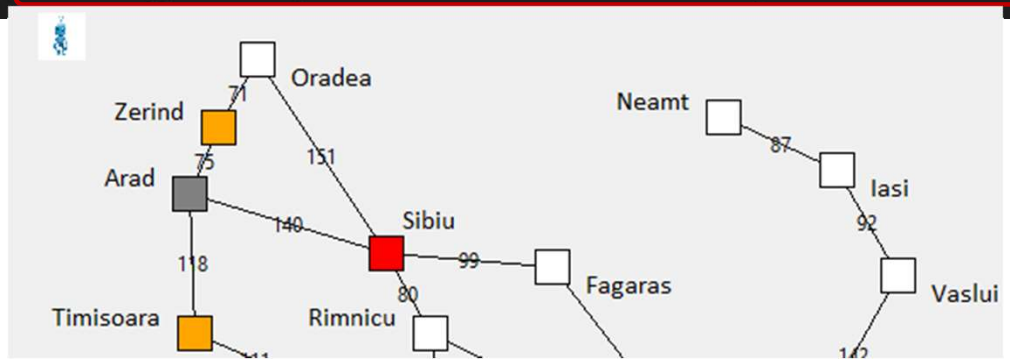
```
search([], [], _) :- fail.
search([(State, _D, IncurredCost, _, Soln) | _], Soln, IncurredCost) :-
                        goal(State).
search([ State | Frontier], Solution, IncurredCost) :-
                      write('Expanding ...  \n'), write(State),write('\n'),
                      expand(State, Nodes_from_State),
                      estimate(State, Nodes_from_State, Candidates),
                      /* write('Candidates: '), write(Candidates),nl,*/
                      insert_all(Candidates, Frontier, NewFrontier),
                      /* write('\nContinuation from: '),nl,
                       mine_print(NewFrontier), write('RecCall...\n\n'),
                      nl,nl,*/
                      search(NewFrontier, Solution, IncurredCost).
```




```
/*
The version of the 'expand' predicate given here simply uses Prolog's bagof computation
(thus bundling up a lot of work).

bagof(+Template, :Goal, -Bag)
*/
expand((State, _, _, _, Ancestors), AllNodeFromState) :-
              bagof(
                 (Child, ArcCost, [Move|Ancestors]),
                 (Child, ArcCost, Move)^move(State, Child, ArcCost, Move),
                 AllNodeFromState
                 ).
```

Prolog

```
estimate(_, [], []) :-!.
estimate((State, Depth, IncurredCost, EC, Ancestors),
             [(Child, ArcCost, [Move|Ancestors]) | OtherChildren],
             [(Child, NDepth, NewIncurredCost, EstimatedCost, [Move|Ancestors]) | AlltheRest] ) :-

            NDepth is Depth + 1,
            \+ inpath(Child, Ancestors),
            f_function_new(Child, IncurredCost, ArcCost, F),
            NewIncurredCost is IncurredCost+ArcCost,
            EstimatedCost is IncurredCost+F,
            !,
            estimate((State, Depth, IncurredCost, EC, Ancestors),
                                OtherChildren, AlltheRest).

estimate((State, Depth, IncurredCost, EC, Ancestors), [_| OtherChildren], AlltheRest ) :-
             !,
             estimate((State, Depth, IncurredCost, EC, Ancestors), OtherChildren, AlltheRest).
```

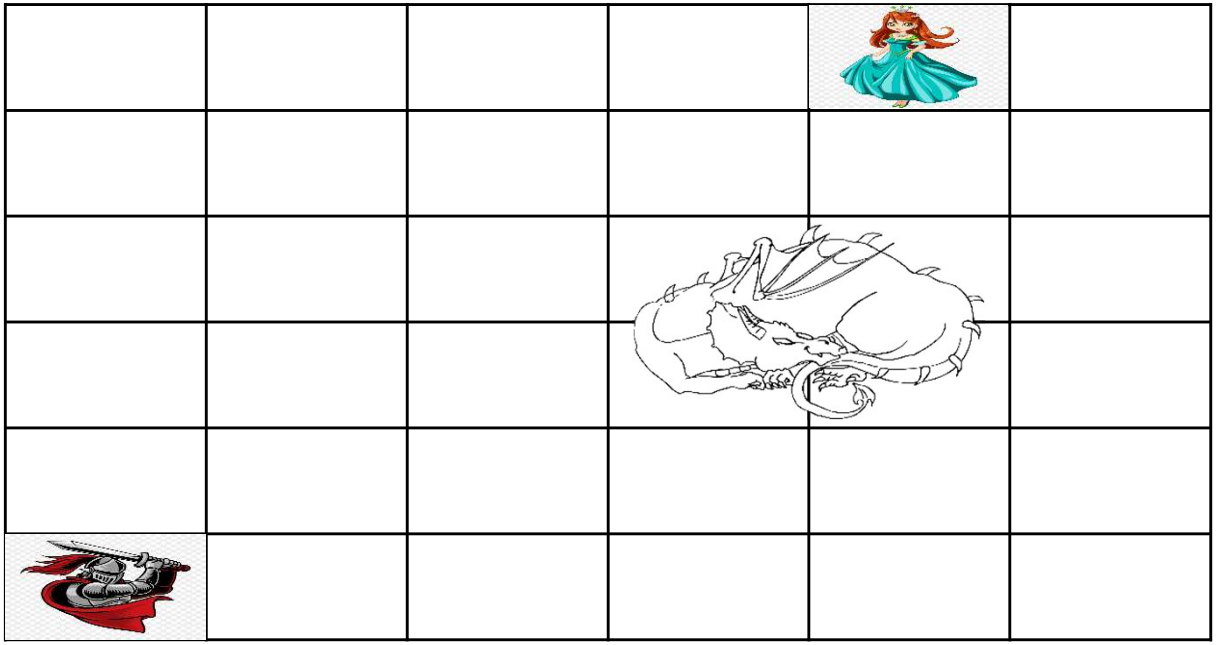


ESEMPI DI ESECUZIONE

Il problema del cavaliere

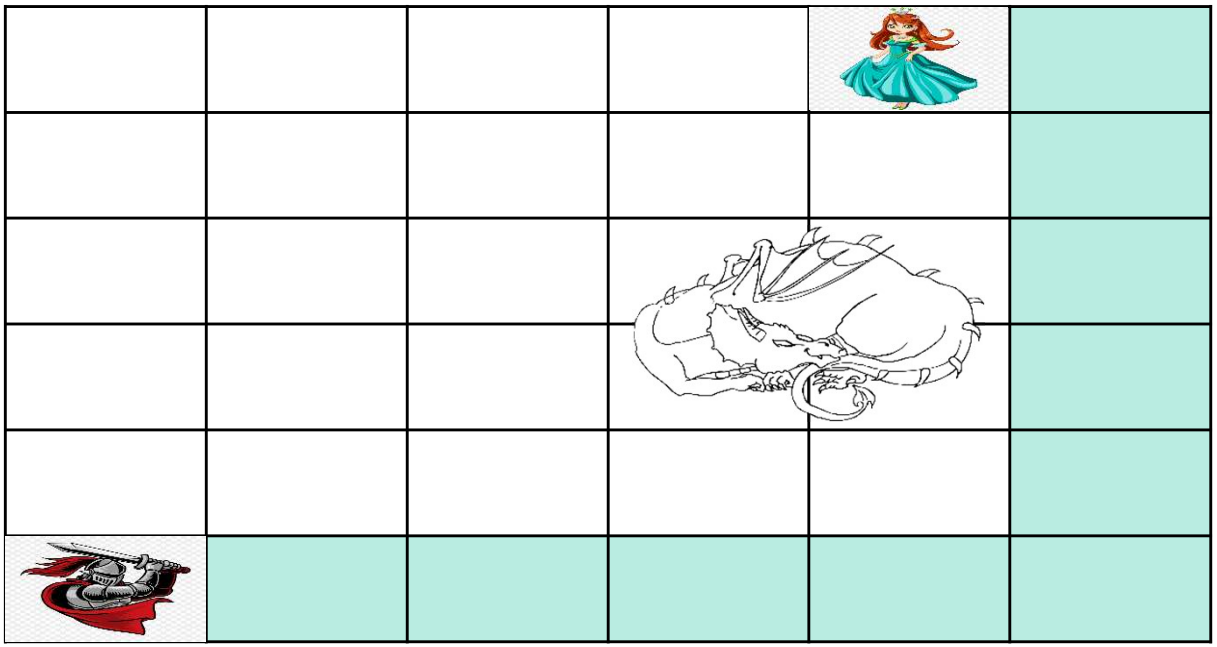
- Una principessa è stata rinchiusa e legata in una grotta insieme ad un drago che le fa da guardia.
- Il cavaliere, che vuole salvarla e riportarla a casa, deve entrare nella grotta, aggirare il drago dormiente, prendere la principessa in braccio e portarla all'uscita.
- Il drago è spietato e non deve essere risvegliato, altrimenti mangerà in un solo boccone il cavaliere.
- Le luci sono accese, c'è una sola entrata/uscita.

Esempio



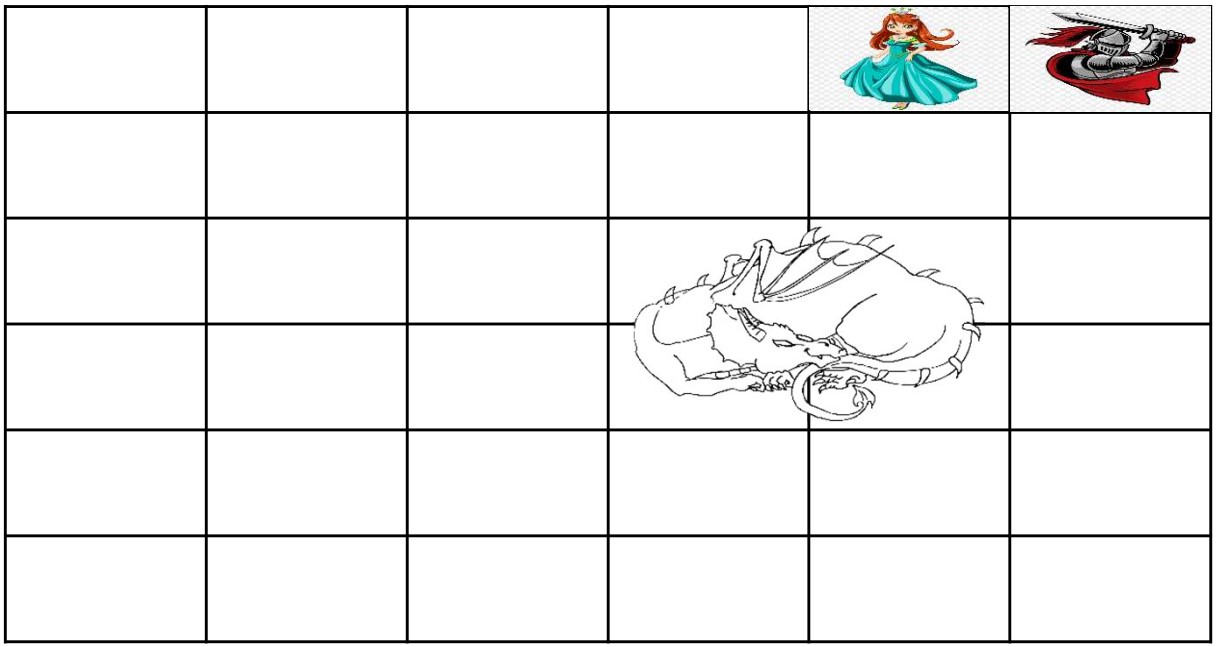
E

Esempio



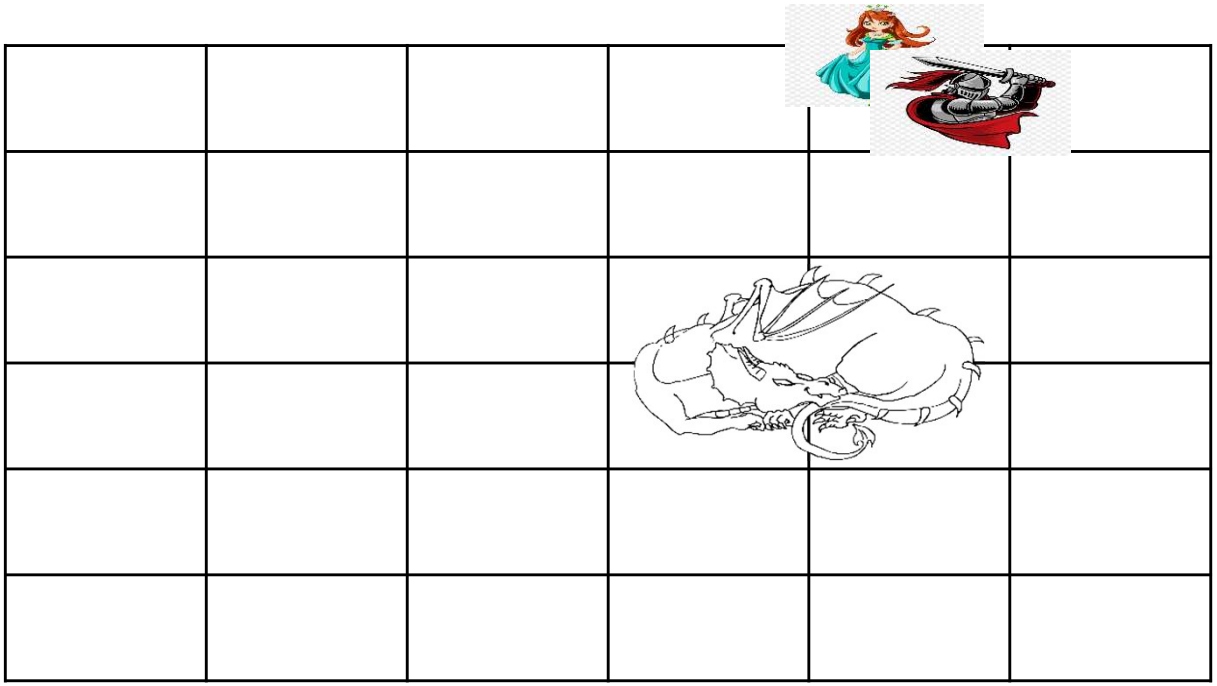
E

Esempio



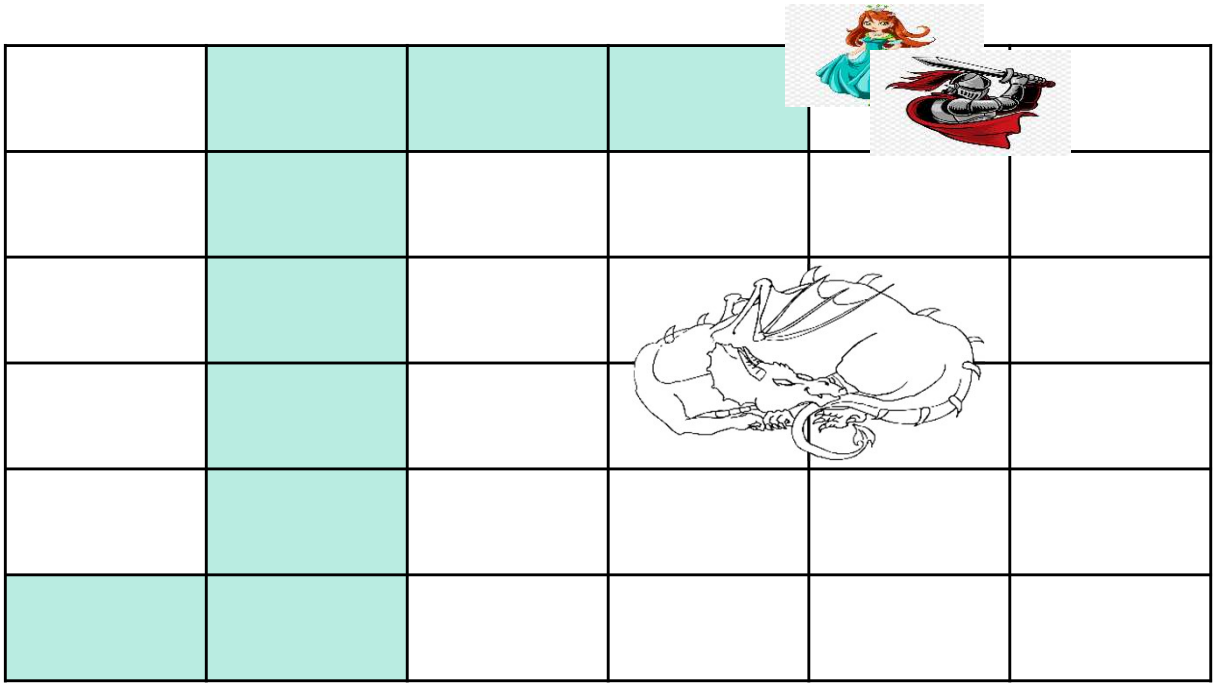
E

Esempio



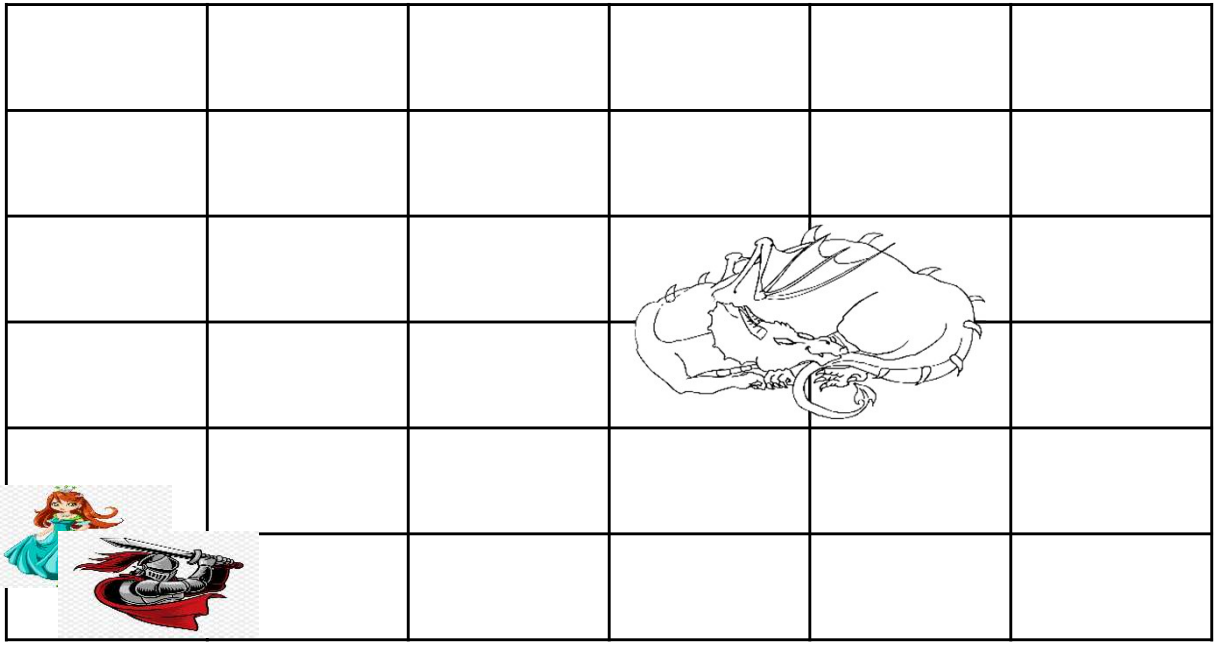
E

Esempio



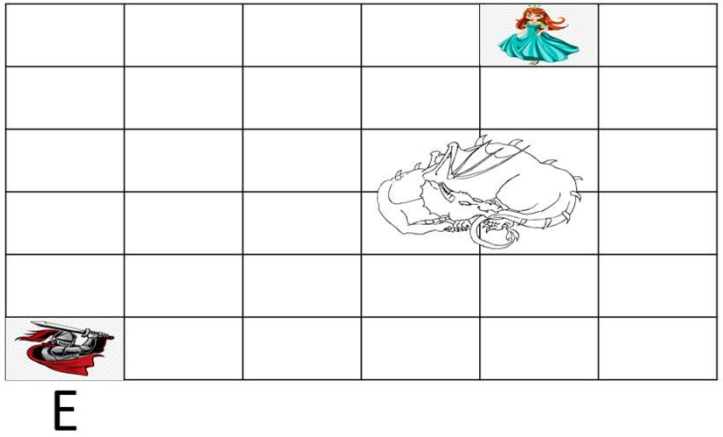
E

Esempio



E

Esercizio in PROLOG!



- Modellare il problema:
  - Definizione *PEAS* dell'agente.
  - Definizione delle proprietà dell'ambiente.
- Dare una soluzione in Prolog, definendo il predicato `save(AgPos, DrPos, ExPos, PrPos, Solution)`, dove:
  - `AgPos` è la posizione (X_a, Y_a) iniziale dell'agente
  - `DrPos` è la posizione $((X_1, Y_1), (X_2, Y_2), (X_3, Y_3), (X_4, Y_4))$ del drago
  - `ExPos` è la posizione di entrata/uscita
  - `PrPos` è la posizione (X_p,Y_p) della principessa
  - `Solution` è la sequenza di posizioni (X_i, Y_i) che il cavaliere deve eseguire per salvare la principessa

What if ..?

1. Le luci fossero spente e il cavaliere avesse una torcia che illumina le caselle adiacenti?
2. Il drago non fosse dormiente, ma cieco? E si muovesse casualmente di 1 casella per volta?
3. Cadessero casualmente dei massi dal soffitto della grotta con una probabilità non nulla ad ogni movimento?
4. La grotta fosse su due livelli e ci fossero pochi punti di sola andata da un livello all'altro?
5. Ci fossero delle trappole su alcune caselle che si attivano a pressione?

ESEMPI DI ESECUZIONE

TOCCA A VOI!

Consegnare in un unico file .zip:

- un pdf con la modellazione
- un file prolog con la soluzione

hromei@ing.uniroma2.it