Lab ABAI

Stefano Di Lena

2025

Contents

1 Uninformed Search

Assumiamo per questa esercitazione un'ambiente: osservabile; discreto; noto; deterministico.

Formulazione del Problema

per definire il problema abbiamo bisogno di:

1. uno stato iniziale;
2. le possibili azioni, utilizziamo una funzione ACTIONS(s);
3. RESULT(s, a), che restituisce lo stato ottenuto dopo l'azione;
4. SUCCESSOR(s), che restituisce tutti i possibili stati che è possibile raggiungere dallo stato corrente;
5. uno stato obiettivo GOAL_TEST(s);
6. il path cost (costo di ogni step).

Il problema sarà implementato su un file che chiameremo *problem.py*

Listing 1: problem.py

```
import copy


class Problem:
    def __init__(self, initial_state, goal_state=None):
        self.initial_state = initial_state
        self.goal_state = goal_state

    def successors(self, state):
        pass

    def actions(self, state):
        pass

    def result(self, state, action):
        pass

    def goal_test(self, state):
        if isinstance(self.goal_state, list):
            return state in self.goal_state
        else:
            return state == self.goal_state

    def cost(self, state, action):
        return 1
```

1.1 Street Problem

Se nel nostro caso vogliamo considerare il problema che consiste nel raggiungere Bari partendo da Trani ed abbiamo a disposizione la seguente mappa:

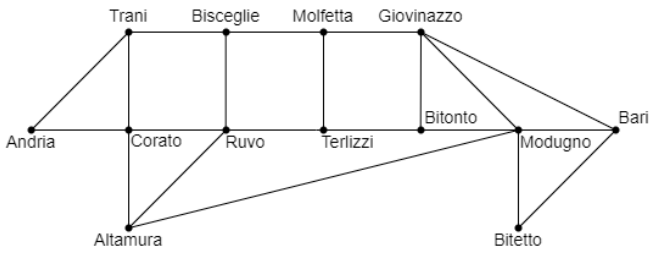

Figure 1: Street Problem

Nel *main.py* andiamo a definire tutti gli stati del problema sotto forma di dizionario [sono in realtà più dizionari annidati (il dizionario principale street ha come chiavi le città che a loro volta sono dizionari avente chiavi le città di destinazione)], di ogni stato conosciamo anche i successori ed il path cost per raggiungerli.

Listing 2: main.py

```
from search.problem import StreetsProblem, EightTilesProblem
from search.tree_search import TreeSearch
from search.graph_search import GraphSearch
from search.strategy import *


streets = {
    'Andria': {'Corato': 3, 'Trani': 2},
    'Corato': {'Andria': 3, 'Ruvo': 2, 'Trani': 3, 'Altamura': 4},
    'Altamura': {'Corato': 4, 'Ruvo': 3, 'Modugno': 5},
    'Ruvo': {'Corato': 2, 'Bisceglie': 3, 'Terlizzi': 2, 'Altamura'
        : 3},
    'Terlizzi': {'Ruvo': 2, 'Molfetta': 2, 'Bitonto': 2},
    'Bisceglie': {'Trani': 2, 'Ruvo': 3, 'Molfetta': 2},
    'Trani': {'Andria': 2, 'Corato': 3, 'Bisceglie': 2},
    'Molfetta': {'Bisceglie': 2, 'Giovinazzo': 2, 'Terlizzi': 2},
    'Giovinazzo': {'Molfetta': 2, 'Modugno': 3, 'Bari': 2, 'Bitonto
        ': 3},
    'Bitonto': {'Modugno': 3, 'Giovinazzo': 3, 'Terlizzi': 2},
    'Modugno': {'Bitonto': 3, 'Giovinazzo': 3, 'Bari': 2, 'Altamura
        ': 5, 'Bitetto': 1},
    'Bari': {'Modugno': 2, 'Giovinazzo': 2, 'Bitetto': 2},
    'Bitetto': {'Bari': 2, 'Modugno': 1}
}
```

Nel file *problem.py* andiamo adesso ad aggiungere in basso la descrizione dello StreetProblem.

Listing 3: problem.py

```
class StreetsProblem:
    def __init__(self, initial_state, goal_state, streets, coords=
        None):
        self.initial_state = initial_state
        self.goal_state = goal_state
        self.streets = streets
        self.coords = coords

    def successors(self, state):
        actions = self.actions(state)
        return [(self.result(state, action), action) for action in
            actions]

    def actions(self, state):
        return self.streets[state].keys()

    def result(self, state, action):
        return action

    def cost(self, state, action):
        return self.streets[state][action]

    def goal_test(self, state):
        return self.goal_state == state
```

Adesso possiamo aggiungere nel main l'implementazione.

Listing 4: main.py

```
problem = StreetsProblem(initial_state = 'Trani', goal_state = '
    Bari', streets = streets)
```

Definire il problema però non basta, dobbiamo cercare una soluzione (una serie di azioni che permetta di raggiungere lo stato obiettivo dallo stato iniziale). Prima però dobbiamo definire la struttura dati Nodo tramite una classe in:

Listing 5: node.py

```
class Node:
    def __init__(self, parent, action, depth, cost, state):
        self.parent = parent
        self.action = action
        self.depth = depth
        self.cost = cost
        self.state = state

    def __repr__(self):
        return str(self.state)

    def expand(self, problem):
        successors = []
        for state, action in problem.successors(self.state):
            successors += [Node(self, action, self.depth+1, self.
                cost+problem.cost(self.state, action), state)]
        return successors

```

```
    def solution(self):
        path = []
        node = self

        while node.parent is not None:
            path.append(node.action)
            node = node.parent

        return path[::-1]
```

Possiamo passare adesso all'implementazione della ricerca.

1.2 Tree Search

Listing 6: tree_search.py

```
from search.node import Node


class TreeSearch:
    def __init__(self, problem, strategy):
        self.problem = problem
        self.strategy = strategy
        self.fringe = []

    def run(self):
        self.fringe.append(Node(None, None, depth=0, cost=0, state=
            self.problem.initial_state))

        while True:
            if len(self.fringe) == 0:
                return 'fail', []

            self.fringe, node = self.strategy.select(self.fringe)

            if self.problem.goal_test(node.state):
                return 'success', node.solution()

            self.fringe += node.expand(self.problem)
```

Se volessimo implementare questo tipo di ricerca nel main scriviamo:

Listing 7: main.py

```
search = TreeSearch(problem, RandomStrategy())
random.seed(100)
print(search.run())
```

Dopo aver implementato però la classe in un nuovo file chiamato *strategy.py*.

Listing 8: strategy.py

```
from random import shuffle


class RandomSearch:
    def select(self, fringe):
```

```
        shuffle(fringe)
        return fringe, fringe.pop(0)
```

1.3 Graph Search

Dato che alcune strategie di ricerca non sono complete a causa di stati ripetuti (possiamo finire in loop) possiamo applicare quest'altra ricerca in cui manteniamo una lista degli stati già esplorati.

```
from search.node import Node


class GraphSearch:
    def __init__(self, problem, strategy):
        self.problem = problem
        self.strategy = strategy
        self.fringe = []
        self.closed = []

    def run(self):
        self.fringe.append(Node(None, None, depth=0, cost=0, state=
            self.problem.initial_state))

        while True:
            if len(self.fringe) == 0:
                return 'fail', []

            self.fringe, node = self.strategy.select(self.fringe)
            if not node:
                return 'fail', []

            if self.problem.goal_test(node.state):
                return 'success', node.solution()

            if node.state not in self.closed:
                self.closed += [node.state]

                fringe_states = [n.state for n in self.fringe]
                self.fringe += [new_node for new_node in node.
                    expand(self.problem) if
                                new_node.state not in fringe_states
                                    ]
```

per runnarla basta modificare nel main:

```
search = GraphSearch(problem, RandomStrategy())
```

1.4 Breadth-Fist Search

L'espansione dei nodi sulla frontiera segue una politica FIFO (si espande prima il root node, poi tutti i successori, poi i successori dei successori...). E' completa se il branching factor "b" è finito. E' ottima se gli step cost sono tutti identici. La complessità spaziale e quella temporale sono entrambe pari ad $O(b^{d+1})$ con "d" la profondità del nodo obiettivo nella soluzione migliore.

Listing 9: strategy.py

```
class BreadthFirstSearch:
    def select(self, fringe):
        return fringe, fringe.pop(0)
```

1.5 Uniform-Cost Search

Si espande per primo il nodo avente path cost minore. E' ottima rispetto al costo in ogni caso (la prima soluzione che trova avrà un costo basso almeno quanto quello di ogni altro nodo sulla frontiera), i cammini sono esaminati sistematicamente in ordine di costo crescente. La completezza è garantita solo se il costo di ogni step è maggiore rispetto ad una costante positiva epsilon e "b" è finito. La complessità spaziale e temporale dell'algoritmo è $O(b^{1+(C*/\epsilon)})$; con "C*" costo della soluzione ottima.

Listing 10: strategy.py

```
class UniformCostSearch:
    def select(self, fringe):
        fringe = sorted(fringe, key=lambda n: n.cost)
        return fringe, fringe.pop(0)
```

1.6 Depth-Fist Search

Si espande direttamente il nodo a profondità maggiore nella frontiera, quando un ramo non può essere più espanso, se non è stato trovato l'obiettivo, viene escluso dalla frontiera e la ricerca ricomincia al successivo nodo più profondo che ha successori non esplorati. E' incompleta. Non è ottima (restituisce la prima soluzione che trova, anche se non è la meno costosa). Complessità temporale $O(b^m)$; "m" massima profondità dell'albero. Complessità spaziale $O(bm)$.

Listing 11: strategy.py

```
class DepthFirstSearch:
    def select(self, fringe):
        return fringe, fringe.pop()
```

1.7 Depth-Limited Search

Imponiamo un limite di profondità "l", in questo modo è risolto il problema dei cicli infiniti, però potrebbe comunque non raggiungere la completezza. Complessità temporale e spaziale sono rispettivamente $O(b^l)$ e $O(bl)$.

Listing 12: strategy.py

```
class DepthLimitedSearch:
    def __init__(self, limit):
        self.limit = limit
    def select(self, fringe):
```

```
        node = fringe.pop()
        if node.depth <= self.limit:
            return fringe, node
        return None, None
```

Queste le possiamo provare ora nel main scrivendo:

```
search = GraphSearch(problem, BreadthFirstSearch())
print(search.run())
search = GraphSearch(problem, UniformCostSearch())
print(search.run())
search = GraphSearch(problem, DepthFirstSearch())
print(search.run())
search = GraphSearch(problem, DepthLimitedSearch(2))
print(search.run())
```

1.8 Eight Tiles Puzzle

Definizione del problema:

Listing 13: problem.py

```
class EightTilesProblem:
    def __init__(self, initial_state, goal_state):
        self.initial_state = initial_state
        self.goal_state = goal_state

    def successors(self, state):
        actions = self.actions(state)
        return [(self.result(state, action), action) for action in
            actions]

    def result(self, state, action):
        new_state = copy.deepcopy(state)
        index = new_state.index(0)
        row = index // 3
        col = index % 3

        new_row, new_col = row, col

        if action == 'up':
            new_row -= 1
        if action == 'down':
            new_row += 1
        if action == 'left':
            new_col -= 1
        if action == 'right':
            new_col += 1

        new_index = new_row * 3 + new_col

        new_state[new_index], new_state[index] = state[index],
            state[new_index]

        return new_state
```

```
32
33      def actions(self, state):
34          index = state.index(0)
35          row = index // 3
36          col = index % 3
37
38          actions = ['up', 'down', 'left', 'right']
39
40          if row < 1:
41              actions.remove('up')
42          if row >= 2:
43              actions.remove('down')
44          if col < 1:
45              actions.remove('left')
46          if col >= 2:
47              actions.remove('right')
48
49          return actions
50
51      def goal_test(self, state):
52          return self.goal_state == state
53
54      def cost(self, state, action):
55          return 1
```

L'implementazione dello stesso avviene nel main scrivendo:

Listing 14: main.py

```
1 problem = EightTilesProblem([1, 2, 3, 4, 0, 5, 6, 7, 8], [1, 2, 3,
      4, 5, 6, 7, 8, 0])
```

2 Informed Search

Considerando sempre lo Street problem:

Listing 15: problem.py

```
import math
import copy


class Problem:
    def __init__(self, initial_state, goal_state=None):
        self.initial_state = initial_state
        self.goal_state = goal_state

    def successors(self, state):
        pass

    def actions(self, state):
        pass

    def result(self, state, action):
        pass

    def goal_test(self, state):
        if isinstance(self.goal_state, list):
            return state in self.goal_state
        else:
            return state == self.goal_state

    def cost(self, state, action):
        return 1

    def heuristic(self, state):
        pass


class StreetsProblem:
    def __init__(self, initial_state, goal_state, streets, coords=
        None):
        self.initial_state = initial_state
        self.goal_state = goal_state
        self.streets = streets
        self.coords = coords

    def successors(self, state):
        actions = self.actions(state)
        return [(self.result(state, action), action) for action in
            actions]

    def actions(self, state):
        return self.streets[state].keys()

    def result(self, state, action):
        return action

    def cost(self, state, action):
        return self.streets[state][action]
```

```
51
52      def goal_test(self, state):
53          return self.goal_state == state
54
55      def heuristic(self, state):
56          x_current, y_current = self.coords[state]
57          x_goal, y_goal = self.coords[self.goal_state]
58
59          return math.sqrt((x_current - x_goal) ** 2 + (y_current -
                y_goal) ** 2)
```

Listing 16: main.py

```
1  from search.problem import StreetsProblem, EightTilesProblem
2  from search.tree_search import TreeSearch
3  from search.graph_search import GraphSearch
4  from search.strategy import *
5
6
7  streets = {
8      'Andria': {'Corato': 3, 'Trani': 2},
9      'Corato': {'Andria': 3, 'Ruvo': 2, 'Trani': 3, 'Altamura': 4},
10     'Altamura': {'Corato': 4, 'Ruvo': 3, 'Modugno': 5},
11     'Ruvo': {'Corato': 2, 'Bisceglie': 3, 'Terlizzi': 2, 'Altamura'
           : 3},
12     'Terlizzi': {'Ruvo': 2, 'Molfetta': 2, 'Bitonto': 2},
13     'Bisceglie': {'Trani': 2, 'Ruvo': 3, 'Molfetta': 2},
14     'Trani': {'Andria': 2, 'Corato': 3, 'Bisceglie': 2},
15     'Molfetta': {'Bisceglie': 2, 'Giovinazzo': 2, 'Terlizzi': 2},
16     'Giovinazzo': {'Molfetta': 2, 'Modugno': 3, 'Bari': 2, 'Bitonto
           ': 3},
17     'Bitonto': {'Modugno': 3, 'Giovinazzo': 3, 'Terlizzi': 2},
18     'Modugno': {'Bitonto': 3, 'Giovinazzo': 3, 'Bari': 2, 'Altamura
           ': 5, 'Bitetto': 1},
19     'Bari': {'Modugno': 2, 'Giovinazzo': 2, 'Bitetto': 2},
20     'Bitetto': {'Bari': 2, 'Modugno': 1}
21 }
22
23 cities_coords = {
24     'Andria': (41.2316, 16.2917),
25     'Corato': (41.1465, 16.4147),
26     'Altamura': (40.8302, 16.5545),
27     'Ruvo': (41.1146, 16.4886),
28     'Terlizzi': (41.1321, 16.5461),
29     'Bisceglie': (41.243, 16.5052),
30     'Trani': (41.2737, 16.4162),
31     'Molfetta': (41.2012, 16.5983),
32     'Giovinazzo': (41.1874, 16.6682),
33     'Bitonto': (41.1118, 16.6902),
34     'Modugno': (41.0984, 16.7788),
35     'Bari': (41.1187, 16.852),
36     'Bitetto': (41.040, 16.748)
37 }
38
39 problem = StreetsProblem(initial_state='Trani', goal_state='Bari',
       streets=streets, coords=cities_coords)
```

2.1 Greedy Search

Viene sfruttata la conoscenza specifica del dominio applicativo per fornire suggerimenti sul dove potrebbe essere l'obiettivo. I suggerimenti hanno la firma di una funzione euristicha "h(n)" costo del cammino più economico dallo stato del nodo n ad uno obiettivo. Quando n è il nodo obiettivo h(n)=0.

É Considerata una funzione di valutazione "f(n) = h(n)" per espandere i nodi, quindi espande prima il nodo che appare più vicino all'obiettivo. La soluzione trovata con questa ricerca non è sempre la migliore (non è ottimale) perché tiene conto solo dell'euristica e non del costo complessivo del cammino. Complessità temporale e spaziale sono $O(b^m)$.

Listing 17: strategy.py

```
class GreedySearch:
    def __init__(self, problem):
        self.problem = problem

    def select(self, fringe):
        fringe = sorted(fringe, key=lambda n: self.problem.
            heuristic(n.state))
        return fringe, fringe.pop(0)
```

2.2 A* Search

Valuta i nodi combinando il costo necessario a raggiungere il nodo ed il costo per andare all'obiettivo "f(n) = g(n) + h(n)". Questa ricerca è completa (in spazi stati finiti), ammissibile (non sovrastima mai il costo per raggiungere un obiettivo) e quindi è ottima rispetto al costo. Per l'ammissibilità deve accadere che "h(n) ¡= h(n)*" dove h* è il costo del cammino ottimo. Un'altra proprietà è la consistenza: "h(n) ¡= c(n,a,n') + h(n')". Questa rappresenta la formula generale della disuguaglianza triangolare, per la quale ogni lato di un triangolo non può essere più lungo della somma degli altri due. [Ogni euristica consistente è ammissibile (non vale il viceversa)].

La complessità spaziale e temporali sono analizzate basandosi su: l'errore assoluto $\Delta = h^* - h$ e l'errore relativo $\epsilon = \frac{h^* - h}{h^*}$. Risulta alla fine pari a $O(b^{\epsilon d})$.

Listing 18: strategy.py

```
class AStarSearch:
    def __init__(self, problem):
        self.problem = problem

    def select(self, fringe):
        fringe = sorted(fringe, key=lambda n: n.cost + self.problem
            .heuristic(n.state))
        return fringe, fringe.pop(0)
```

Possiamo implementare le ricerche nel main così:

```
search = GraphSearch(problem, GreedySearch(problem))
print(search.run())
```

```
3
4 search = GraphSearch(problem, AStarSearch(problem))
5 print(search.run())
```

2.3 Tiles Puzzle

Listing 19: problem.py

```
1  class EightTilesProblem:
2      def __init__(self, initial_state, goal_state):
3          self.initial_state = initial_state
4          self.goal_state = goal_state
5
6      def successors(self, state):
7          actions = self.actions(state)
8          return [(self.result(state, action), action) for action in
               actions]
9
10     def result(self, state, action):
11         new_state = copy.deepcopy(state)
12         index = new_state.index(0)
13         row = index // 3
14         col = index % 3
15
16         new_row, new_col = row, col
17
18         if action == 'up':
19             new_row -= 1
20         if action == 'down':
21             new_row += 1
22         if action == 'left':
23             new_col -= 1
24         if action == 'right':
25             new_col += 1
26
27         new_index = new_row * 3 + new_col
28
29         new_state[new_index], new_state[index] = state[index],
               state[new_index]
30
31         return new_state
32
33     def actions(self, state):
34         index = state.index(0)
35         row = index // 3
36         col = index % 3
37
38         actions = ['up', 'down', 'left', 'right']
39
40         if row < 1:
41             actions.remove('up')
42         if row >= 2:
43             actions.remove('down')
44         if col < 1:
45             actions.remove('left')
```

```
        if col >= 2:
            actions.remove('right')

        return actions

    def goal_test(self, state):
        return self.goal_state == state

    def cost(self, state, action):
        return 1

    """
    def heuristic(self, state):
        # number of misplaced tiles
        return sum([x != y for x, y in zip(state, self.goal_state)
            if x != 0])

    """

    def heuristic(self, state):
        # total manhattan distance
        distance = 0
        for tile in range(1, 9):
            current_index = state.index(tile)
            current_row = current_index // 3
            current_col = current_index % 3
            goal_index = self.goal_state.index(tile)
            goal_row = goal_index // 3
            goal_col = goal_index % 3
            distance += abs(current_row - goal_row) + abs(
                current_col - goal_col)

        return distance
```

Possiamo utilizzare i due tipi di euristica a seconda delle esigenze (basta invertire i commenti in questo caso per usare una invece di un'altra) o scrivere direttamente solo quella che ci interessa.

Nel main implementiamo il problema così:

Listing 20: main.py

```
problem = EightTilesProblem([7, 2, 4, 5, 0, 6, 8, 3, 1], [1, 2, 3,
    4, 5, 6, 7, 8, 0])
# testare l'euristica
print(problem.heuristic([7, 2, 4, 5, 0, 6, 8, 3, 1]))


# risolvere il problema nella stessa configurazione usata nella
    ricerca non informata
problem = EightTilesProblem([1, 2, 3, 4, 0, 5, 6, 7, 8], [1, 2, 3,
    4, 5, 6, 7, 8, 0])
search = GraphSearch(problem, GreedySearch(problem))
print(search.run())

search = GraphSearch(problem, AStarSearch(problem))
print(search.run())
```

3 Local Search

Non si tiene traccia dei cammini o degli stati già raggiunti. Non sono sistematici. Utilizzano però poca memoria e trovano soluzioni ragionevoli anche in spazi di stati infiniti.

3.1 Queen Problem

Listing 21: problem.py

```
class EightQueenProblem:
    def __init__(self, initial_state, goal_state=None):
        self.initial_state = initial_state
        self.goal_state = goal_state

    def successors(self, state):
        actions = self.actions(state)
        return [(self.result(state, action), action) for action in
            actions]

    def actions(self, state):
        actions = []
        for col, row in enumerate(state):
            actions += [(col, new_row) for new_row in range(8) if
                new_row != row]
        return actions

    def result(self, state, action):
        new_state = state[:]
        col, row = action
        new_state[col] = row
        return new_state

    def goal_test(self, state):
        return self.heuristic(state) == 0

    def cost(self, state, action):
        return 1

    def heuristic(self, state):
        conflict = 0

        for col1 in range(8):
            row1 = state[col1]
            for col2 in range(col1+1, 8):
                row2 = state[col2]
                if row1 == row2:
                    conflict += 1
                if abs(row1 - row2) == abs(col1 - col2):
                    conflict += 1

        return conflict

    def value(self, state):
        return - self.heuristic(state)
```

Questo problema consiste nel piazzare 8 regine su una scacchiera in modo tale che non risultano adiacenti tra di loro.

Listing 22: main.py

```
from search.problem import EightQueenProblem
from search.local_search import HillClimbingSearch,
    SimulatedAnnealingSearch, GeneticAlgorithm
import random

initial = [random.randint(0, 7) for _ in range(8)]
problem = EightQueenProblem(initial)
```

3.2 Hill Climbing

Iniziando da uno stato iniziale (casuale) ad ogni iterazione passa allo stato vicino che punta nella direzione che presenta l'ascesa più ripida. Potrebbe capitare che questo si blocchi a causa di: minimo locale, ridges (creste), Platteaux (spalla).

Listing 23: local_search.py

```
from search.node import Node
import random
import math


class HillClimbingSearch:
    def __init__(self, problem):
        self.problem = problem

    def run(self):
        current = Node(state=self.problem.initial_state, parent=
            None, cost=0, depth=0, action=None)

        while True:
            neighbors = current.expand(self.problem)
            best = min(neighbors, key=lambda n: self.problem.
                heuristic(n.state))

            if self.problem.heuristic(best.state) >= self.problem.
                heuristic(current.state):
                return current, self.problem.heuristic(current.
                    state)

            current = best
```

Listing 24: main.py

```
search = HillClimbing(problem)
print(search.run())
```

Il successo di questo algoritmo dipende dallo spazio degli stati, per questo spesso vengono effettuate più ricerche con generazione casuale dello stato iniziale.

Listing 25: main.py

```
it = 0
while True:
    it+=1
    search = HillClimbingSearch(problem=EightQueenProblem([random.
        randint(0, 7) for _ in range(8)]))
    result = search.run()
    if result[1] == 0:
        print(f"Found Solution {result[0].state}, with cost {result
            [1]}, after {it} iterations.")
        break
```

In questo caso non usiamo le variabili *problem* ed *initial* a priori (ma sono istanziate in linea nell'algoritmo di ricerca) così da rendere lo stato iniziale casuale ad ogni istanza dell'algoritmo.

3.3 Simulated Annealing

Permette all'hill climbing di effettuare delle mosse errate decrementandone gradualmente la dimensione e frequenza. [probabilità concessa per mosse errate: $p(x) = \alpha e^{E(x)/kT}$].

Listing 26: local_search.py

```
class SimulatedAnnealingSearch:
    def __init__(self, problem, max_time, schedule):
        self.problem = problem
        self.max_time = max_time
        self.schedule = schedule

    def run(self):
        current = Node(state=self.problem.initial_state, parent=
            None, depth=0, action=None, cost=0)

        for time in range(self.max_time):
            temp = self.schedule(time)

            if temp == 0:
                return current

            neighbour = random.choice(current.expand(self.problem))

            delta = self.problem.heuristic(current.state) - self.
                problem.heuristic(neighbour.state)

            if delta > 0 or random.uniform(0, 1) < math.exp(delta/
                temp):
                current = neighbour
                print(current, self.problem.heuristic(current.state
                    ))

                if self.problem.heuristic(current.state) == 0:
                    break

        return current, self.problem.heuristic(current.state)
```

É implementato così:

Listing 27: main.py

```
def scheduler(time):
    initial_temp = 1000
    lamb = 0.001
    return initial_temp - lamb * time

search = SimulatedAnnealingSearch(problem=problem, max_time=100000,
     schedule=scheduler)
print(search.run())
```

3.4 Genetic Algorithms

Inizia con un set di k stati generati casualmente (popolazione). Ogni stato è rappresentato da una stringa e valutato da una fitness function (assegna valori maggiori ai migliori stati). Due coppie vengono scelte casualmente per la riproduzione. Un punto di crossover è scelto casualmente nella stringa. I successori sono creati ricombinando le parti separate dai punti di incrocio (prima parte appartiene al genitore 1 e l'altra al 2). Ogni prole poi potrebbe essere soggetta a mutazioni casuali.

Listing 28: local_search.py

```
class GeneticAlgorithm:
    def __init__(self, problem, population_size, max_generation,
        state_len, gene_pool, mutation_rate):
        self.problem = problem
        self.gene_pool = gene_pool
        self.mutation_rate = mutation_rate
        self.population_size = population_size
        self.state_len = state_len
        self.max_generation = max_generation

    def select(self, population):
        fitnesses = [1 / (1 + self.problem.heuristic(individual))
            for individual in population]
        probability = [fitness / sum(fitnesses) for fitness in
            fitnesses]
        return random.choices(population, weights=probability, k=2)

    def crossover(self, parent1, parent2):
        c_point = random.randint(0, self.state_len)
        return parent1[:c_point] + parent2[c_point:]

    def mutation(self, individual):
        if random.uniform(0, 1) < self.mutation_rate:
            pos = random.choice(range(self.state_len))
            individual[pos] = random.choice(self.gene_pool)
        return individual

    def run(self):
        population = [random.sample(self.gene_pool, k=self.
            state_len) for _ in range(self.population_size)]
```

```
        best = None

        for _ in range(self.max_generation):
            population = [self.mutation(self.crossover(*self.select
                (population))) for _ in range(self.population_size)
                ]
            best = max(population, key=self.problem.value)
            print(best, self.problem.value(best))

            if self.problem.goal_test(best):
                break

        return best, self.problem.value(best)
```

L'implementazione è la seguente:

Listing 29: main.py

```
search = GeneticAlgorithm(problem=problem,
                          population_size=100,
                          max_generation=1000,
                          state_len=8,
                          gene_pool=[i for i in range(8)],
                          mutation_rate=0.3)

print(search.run())
```

4 CSP

Iniziamo creando una classe in un programma a parte, che richiameremo (perché la struttura del csp è fissa).

Listing 30: csp.py

```
class CSP:
    def __init__(self, variables, domains, constraints):
        self.variables = variables
        self.domains = domains
        self.constraints = constraints

    def consistent(self, assignment):
        return all(constraint.check(assignment) for constraint in
            self.constraints)

    def complete(self, assignment):
        return len(assignment) == len(self.variables)

    def assign(self, assignment, variable, value):
        assignment[variable] = value
        return assignment

    def unassign(self, assignment, variable):
        assignment.pop(variable)
        return assignment
```

4.1 Map Coloring

Listing 31: main.py

```
from csp.csp import CSP
from csp.backtrack import Backtracking, BacktrackingFC
from csp.heuristics import *
from csp.arc_consistency import AC3
from csp.local import MinConflict


class DifferentValues:
    def __init__(self, var1, var2):
        self.var1 = var1
        self.var2 = var2

    def check(self, assignment):
        value1 = assignment.get(self.var1)
        value2 = assignment.get(self.var2)
        if value1 and value2:
            return value1 != value2
        return True


variables = ['WA', 'NT', 'Q', 'NSW', 'V', 'SA', 'T']
domain = ['red', 'green', 'blue']
domains = {variable: domain for variable in variables}
```

```
constraints = [
    DifferentValues('SA', 'WA'),
    DifferentValues('SA', 'NT'),
    DifferentValues('SA', 'Q'),
    DifferentValues('SA', 'NSW'),
    DifferentValues('SA', 'V'),
    DifferentValues('WA', 'NT'),
    DifferentValues('NT', 'Q'),
    DifferentValues('Q', 'NSW'),
    DifferentValues('NSW', 'V')
]


problem = CSP(variables=variables, domains=domains, constraints=
    constraints)
```

4.2 Backtracking Search

Si sceglie una variabile, si prova ad associare un valore ad essa, se è inconsistente viene fatto backtrack.

Listing 32: backtrack.py

```
import copy


class Backtracking:
    def __init__(self, csp, variable_criterion, value_criterion):
        self.csp = csp
        self.variable_criterion = variable_criterion
        self.value_criterion = value_criterion

    def backtrack_search(self, assignment):
        if self.csp.complete(assignment) and self.csp.consistent(
            assignment):
            return assignment

        variable = self.variable_criterion(self.csp, assignment)

        for value in self.value_criterion(self.csp, variable):
            self.csp.assign(assignment, value=value, variable=
                variable)

            if self.csp.consistent(assignment):
                result = self.backtrack_search(assignment)

                if result:
                    return result

            self.csp.unassign(assignment, variable=variable)

        return False

    def run(self):
        return self.backtrack_search(assignment={})
```

Si implementa così:

Listing 33: main.py

```
search = Backtracking(problem, variable_criterion=random_variable,
    value_criterion=unordered_value)
print(search.run())
```

Ma prima bisogna implementare due euristiche necessarie a selezionare i valori e le variabili.

Listing 34: heuristics.py

```
import random

def random_variable(csp, assignment):
    return random.choice([var for var in csp.variables if var not
        in assignment.keys()])

def unordered_value(csp, variable):
    return csp.domains[variable]
```

4.2.1 Heuristics

Altre euristiche utilizzabili che potrebbero incrementare l'efficienza della ricerca sono: MRV, Degree.

Listing 35: heuristics.py

```
def minimum_remaining_values(csp, assignment):
    unassigned = [var for var in csp.variables if var not in
        assignment.keys()]
    return min(unassigned, key = lambda v: len(csp.domains[v]))


def degree_heuristic(csp, assignment):
    unassigned = [var for var in csp.variables if var not in
        assignment.keys()]
    constraints_count = {v:0 for v in unassigned}

    for c in csp.constraints:
        if c.var1 in unassigned and c.var2 in unassigned:
            constraints_count[c.var1] += 1
            constraints_count[c.var2] += 1

    return max(unassigned, key = lambda v: constraints_count[v])
```

Listing 36: main.py

```
search = Backtracking(problem, variable_criterion=
    minimum_remaining_values, value_criterion=unordered_value)
print(search.run())

search = Backtracking(problem, variable_criterion=degree_heuristic,
     value_criterion=unordered_value)
print(search.run())
```

4.3 Inference

Usando la constrain propagation dobbiamo modificare il file *csp.py*.

Listing 37: csp.py

```
class CSP:
    def __init__(self, variables, domains, constraints):
        self.variables = variables
        self.domains = domains
        self.constraints = constraints
        self.neighbours = self.compute_neighbours()

    def compute_neighbours(self):
        neighbours = {}
        for c in self.constraints:
            neighbours.setdefault(c.var1, []).append(c.var2)
            neighbours.setdefault(c.var2, []).append(c.var1)
        return neighbours


    def consistent(self, assignment):
        return all(constraint.check(assignment) for constraint in
            self.constraints)

    def complete(self, assignment):
        return len(assignment) == len(self.variables)

    def assign(self, assignment, variable, value):
        assignment[variable] = value
        return assignment

    def unassign(self, assignment, variable):
        assignment.pop(variable)
        return assignment
```

4.4 AC3

Listing 38: arc_consistency.py

```
class AC3:
    def __init__(self, csp):
        self.csp = csp
        self.queue = self.arcs()


    def arcs(self):
        queue = []

        for var in self.csp.neighbours.keys():
            queue += [(var, neighbour) for neighbour in self.csp.
                neighbours[var]]

        return queue

    def remove_inconsistent_values(self, var1, var2):
```

```
        removed = False
        for value1 in self.csp.domains[var1][:]:
            constrains_results = [self.csp.consistent({var1: value1
                , var2: value2}) for value2 in self.csp.domains[
                var2]] #for constraint in self.csp.constraints if
                constraint.var1 == var1 and constraint.var2 == var2
                ]
            if not any(constrains_results):
                self.csp.domains[var1].remove(value1)
                removed = True

        return removed

    def run(self):
        while len(self.queue)>0:
            var1, var2 = self.queue.pop(0)

            if self.remove_inconsistent_values(var1, var2):

                if len(self.csp.domains[var1]) == 0:
                    return False

                for var_n in self.csp.neighbours[var1]:
                    if var_n != var2:
                        self.queue.append((var_n, var1))

        return True
```

Listing 39: main.py

```
ac3 = AC3(problem)
print(ac3.run())

# viasualizzare i domini dopo l'arco consistenza
print(ac3.csp.domains)
```

4.5 Forward Checking

Listing 40: backtrack.py

```
class BacktrackingFC:
    def __init__(self, csp, variable_criterion, value_criterion):
        self.csp = csp
        self.variable_criterion = variable_criterion
        self.value_criterion = value_criterion

    def forward_check(self, assignment, variable):
        new_domains = copy.deepcopy(self.csp.domains)

        for var in self.csp.variables:
            new_assignment = copy.deepcopy(assignment)
            if var not in new_assignment and var in self.csp.
                neighbours.get(variable, []):
                for value in new_domains[var][:]:
                    self.csp.assign(new_assignment, var, value)
```

```
                    if not self.csp.consistent(new_assignment):
                        new_domains[var] = [v for v in new_domains[
                            var] if v != value]

                if len(new_domains[var]) == 0:
                    return False
        return new_domains

    def backtrack_search(self, assignment):
        if self.csp.complete(assignment) and self.csp.consistent(
            assignment):
            return assignment

        variable = self.variable_criterion(self.csp, assignment)

        for value in self.value_criterion(self.csp, variable):
            self.csp.assign(assignment, value=value, variable=
                variable)

            old_domains = copy.deepcopy(self.csp.domains)
            new_domains = self.forward_check(assignment, variable)

            if new_domains and self.csp.consistent(assignment):
                self.csp.domains = new_domains
                result = self.backtrack_search(assignment)

                if result:
                    return result

            self.csp.domains = old_domains
            self.csp.unassign(assignment, variable=variable)

        return False

    def run(self):
        return self.backtrack_search(assignment={})
```

Listing 41: main.py

```
search = BacktrackingFC(problem, variable_criterion=random_variable
    , value_criterion=unordered_value)
print(search.run())
```

4.6 Local Search

Listing 42: local.py

```
import random

class MinConflict:
    def __init__(self, csp, max_steps):
        self.csp = csp
        self.max_steps = max_steps

    def n_conflicts(self, current, variable, value):
```

```
        count = 0
        for constraint in self.csp.constraints:
            if (variable in {constraint.var1, constraint.var2} and
                    not constraint.check({**current, variable:
                        value})):
                count += 1
        return count

    def run(self):
        current = {var: random.choice(self.csp.domains[var]) for
            var in self.csp.variables}

        for step in range(self.max_steps):
            if self.csp.consistent(current):
                return current

            variable = random.choice([var for var in self.csp.
                variables
                                      if self.n_conflicts(current,
                                          var, current[var]) > 0])

            value = min(self.csp.domains[variable],
                        key=lambda val: self.n_conflicts(current,
                            variable, val))

            current[variable] = value

        return False
```

Listing 43: main.py

```
search = MinConflict(problem, 100)
print(search.run())
```

5 Adversarial Search

Consideriamo adesso unb ambiente con più di un agente, in competizione tra loro.

Listing 44: game.py

```
class Game:
    def __init__(self, initial, terminal, environment):
        self.initial = initial
        self.terminal = terminal
        self.environment = environment
        self.player = 'MAX'

    def actions(self, state):
        return self.environment[state].keys()

    def successors(self, state):
        actions = self.actions(state)
        return [(self.result(state, action), action) for action in
            actions]

    def result(self, state, action):
        return self.environment[state][action]

    def terminal_test(self, state):
        return state in self.terminal.keys()

    def utility(self, state):
        if self.player == 'MAX':
            return self.terminal[state]
        elif self.player == 'MIN':
            return - self.terminal[state]

    def next_player(self):
        if self.player == 'MAX':
            self.player = 'MIN'
        elif self.player == 'MIN':
            self.player = 'MAX'
```

Listing 45: main.py

```
from game.game import Game, TicTacToe
from game.minimax import Minimax
from game.alphabeta import AlphaBeta

dummy_environment = {
    'A': {'a1': 'B', 'a2': 'C', 'a3': 'D'},
    'B': {'b1': 'E', 'b2': 'F', 'b3': 'G'},
    'C': {'c1': 'H', 'c2': 'I', 'c3': 'L'},
    'D': {'d1': 'M', 'd2': 'N', 'd3': 'O'},
}

terminal_state = {
    'E': 3,
    'F': 12,
    'G': 8,
```

```
    'H': 2,
    'I': 4,
    'L': 6,
    'M': 14,
    'N': 5,
    'O': 2
}


game = Game(initial='A', environment=dummy_environment, terminal=
    terminal_state)
```

5.1 Minimax

Listing 46: minimax.py

```
import numpy as np


class Minimax:
    def __init__(self, game):
        self.game = game

    def minimax_decision(self, state):
        return max(self.game.actions(state),
                   key=lambda a: self.min_value(self.game.result(
                       state, a)))

    def min_value(self, state):
        if self.game.terminal_test(state):
            return self.game.utility(state)

        v = np.inf

        for s, a in self.game.successors(state):
            v = min(v, self.max_value(s))
        return v

    def max_value(self, state):
        if self.game.terminal_test(state):
            return self.game.utility(state)

        v = - np.inf

        for s, a in self.game.successors(state):
            v = max(v, self.min_value(s))
        return v

    def run(self):
        moves = []
        state = self.game.initial

        while True:
            if self.game.terminal_test(state):
                return moves
```

```
39
40              action = self.minimax_decision(state)
41              state = self.game.result(state, action)
42              moves.append((self.game.player, action))
43              self.game.next_player()
```

Listing 47: main.py

```
1 search = Minimax(game)
2 print(search.run())
```

5.2 Pruning alfa-beta

Il taglio avviene per un nodo MAX quando il valore corrente è $\geq \beta$; mentre per un nodo MIN quando questo è $\leq \alpha$.

Dove:

- **alfa** è il valore migliore trovato finora per MAX (quello più alto).
- **beta** è il valore migliore trovato finora per MIN (quello più basso).

Listing 48: alphabeta.py

```
1  import numpy as np
2
3
4  class AlphaBeta:
5      def __init__(self, game):
6          self.game = game
7
8      def alphabeta_decision(self, state):
9          return max(self.game.actions(state),
10                     key=lambda a: self.min_value(self.game.result(
                           state, a), -np.inf, np.inf))
11
12     def min_value(self, state, alpha, beta):
13         if self.game.terminal_test(state):
14             return self.game.utility(state)
15
16         v = np.inf
17
18         for s, a in self.game.successors(state):
19             v = min(v, self.max_value(s, alpha, beta))
20             if v <= alpha:
21                 return v
22             beta = min(v, beta)
23
24         return v
25
26     def max_value(self, state, alpha, beta):
27         if self.game.terminal_test(state):
28             return self.game.utility(state)
29
30         v = - np.inf
```

```
31
32          for s, a in self.game.successors(state):
33              v = max(v, self.min_value(s, alpha, beta))
34              if v >= beta:
35                  return v
36              alpha = max(v, alpha)
37          return v
38
39      def run(self):
40          moves = []
41          state = self.game.initial
42
43          while True:
44              if self.game.terminal_test(state):
45                  return moves
46
47              action = self.alphabeta_decision(state)
48              state = self.game.result(state, action)
49              moves.append((self.game.player, action))
50              self.game.next_player()
```

```
1 search = AlphaBeta(game)
2 print(search.run())
```

5.3 Tic Tac Toe

Implementazione del gioco del tris.

Listing 49: game.py

```
1  class TicTacToe:
2      def __init__(self, size):
3          self.size = size
4          self.initial = [' '] * (size * size)
5          self.player = 'X'
6
7      def actions(self, state):
8          return [i for i, cell in enumerate(state) if cell == ' ']
9
10     def successors(self, state):
11         actions = self.actions(state)
12         return [(self.result(state, action), action) for action in
               actions]
13
14     def result(self, state, action):
15         new_state = state.copy()
16         new_state[action] = self.player
17         return new_state
18
19     def check_winner(self, state):
20
21         for i in range(0, len(state), self.size + 1):
22             row = state[i: i+self.size]
23             if ' ' not in row and len(set(row)) == 1:
24                 return row[0]
25
```

```
        for i in range(self.size):
            col = [state[row * self.size + i] for row in range(self
                .size)]
            if ' ' not in col and len(set(col)) == 1:
                return col[0]

        diag = state[0: self.size**2: self.size]
        if ' ' not in diag and len(set(diag)) == 1:
            return diag[0]

        anti_diag = state[self.size-1: self.size**2 - 1: self.size
            - 1]
        if ' ' not in anti_diag and len(set(anti_diag)) == 1:
            return anti_diag[0]

        return None

    def terminal_test(self, state):
        return self.check_winner(state) or ' ' not in state

    def utility(self, state):
        if self.terminal_test(state):
            if self.player == 'X':
                return 1
            elif self.player == 'O':
                return -1
        return 0

    def next_player(self):
        if self.player == 'X':
            self.player = 'O'
        elif self.player == 'O':
            self.player = 'X'
```

Listing 50: main.py

```
game = TicTacToe(size=3)
```