Artificial Intelligence - Notes

Matteo Alberici

January 2022

Contents

1 Search Problems

In order to define a **search problem**, we need to introduce the following terms:

- **States**: the set of possible configurations
- **Initial state**: the state where the search starts
- **Actions**: the set of possible actions
- **Goal**: the destination of the search
- **Cost function**: gives a cost to the solution path

1.1 Graph Definition

A search problem is modelled as a **graph** having the following finite sets:

$$\textbf{nodes}\text{: } (n_1, n_2, ..., n_n)$$
$$\textbf{edges}\text{: } ((n_i, n_j), ..., (n_k, n_n))$$

Given a direct edge (n_i, n_j), we say that n_j is a **child** of N_i. If a node has no children, then it is a **leaf**.
A **path** $(n_i, n_{i+1}, ..., n_n)$ is a sequence with each couple n_i, n_{i+1} being an edge. If a path contains the same node more times, then it is a **cycle**.
A graph is **rooted** when a node n_i is the origin of all the paths. Moreover, two nodes are **connected** if there exists a path from one to the other.

1.2 Search Algorithms

Search algorithms take as input a state space and a starting state, then tries to compute a path producing a search tree. The strategy consists of searching which node to add to the **fringe** and **expand**, considering all nodes reachable in one action from the current one.

1.3 Search Tree

A node in the **search tree** has five components:

- The state
- The node that generated it
- The action that generates it
- The depth of the tree
- The cost of the path from the root

1.4 General Search Algorithm

The following is the procedure of a **general search algorithm** which takes as input a problem and a strategy and returns a sequence of operators that brings to the goal:

1. Initialize the search tree
2. Push the root node to the fringe
3. Start the loop
4. If the fringe is empty, then return failure
5. Choose the next node according to a strategy
6. If the node is the goal, then return the solution
7. Expand the node
8. Add the children to the fringe

1.5 Evaluation of Search Strategies

The strategy determines the order of expansion and is evaluated according to the following criteria:

- **Completeness**: does it always find a solution if it exists?
- **Space Complexity**: how much memory does it require?
- **Time Complexity**: how long does it take to find a solution?
- **Optimality**: does it guarantee the least-cost solution?

Time and space complexities are measured in terms of the maximum branching factor b, the depth of the least-cost solution d, and the maximum depth of the tree m.
Polynomial-time algorithms solve problems in a time growing polynomially with the input size. If such an algorithm does not solve the problem, then the problem is a **nondeterministic-polynomial time problem**.

2 Blind Search Algorithms

Blind search algorithms use non informed search strategies.

2.1 Breadth-first Search Algorithm

The **breadth-first search algorithm** expands the shallowest unexpanded node using a **FIFO** queue: nodes at distance d are expanded before nodes at distance $d + 1$.
The procedure of the algorithm is the following:

1. Push the root node to the fringe
2. If the fringe is empty, then return failure
3. Remove the first node n from the fringe
4. If n is the goal, then return the solution
5. If n is in the explored list, then repeat from step 2
6. Expand n
7. Add the children at the end of the fringe
8. Add n to the explored list
9. Repeat from step 2

The algorithm is evaluated as follows:

- Completeness: yes
- Space Complexity: $O(b^d)$ - all nodes until goal depth are created
- Time Complexity: $O(b^d)$ - all nodes until goal depth are created
- Optimality: yes, if all steps have the same cost $g(n) = k \cdot depth(n)$

2.2 Uniform Cost Search Algorithm

The **uniform cost search algorithm** associates a cost $g(n)$ to each node and expands the one with the lowest $g(n)$ in the fringe, which is kept sorted in increasing cost order.
The procedure of the algorithm is the following:

1. Push the root node to the fringe
2. If the fringe is empty, then return failure
3. Remove the first node n from the fringe
4. If n is the goal, then return the solution
5. If n is in the explored list, then repeat from step 2
6. Expand n
7. Add the children not in the explored list to the fringe
8. Sort the fringe by the path cost
9. Add n to the explored list
10. Repeat from step 2

If $g(n)$ is equal to the depth of node n, then the procedure is the same of the breadth-first search algorithm.
The algorithm is evaluated as follows:

- Completeness: yes, if the step cost $\geq \epsilon > 0$
- Space Complexity: $O(b^d)$, number of nodes with $g(n) \leq g(\text{solution})$
- Time Complexity: $O(b^d)$, number of nodes with $g(n) \leq g(\text{solution})$
- Optimality: yes, if all costs are non-negative

2.3 Depth-first Search Algorithm

The **depth-first search algorithm** expands a node generated in the previous step using a **LIFO** queue.
The procedure of the algorithm is the following:

1. Push the root node to the fringe
2. If the fringe is empty, then return failure
3. Remove the first node n from the fringe
4. If n is the goal, then return the solution
5. If n is in the explored list, then repeat from step 2
6. Expand n
7. Add the children not in the explored list to the beginning of the fringe
8. Add n to the explored list
9. Repeat from step 2

The algorithm is evaluated as follows:

- Completeness: no, it could proceed to infinity
- Space Complexity: $O(b \cdot m)$
- Time Complexity: $O(b^m)$
- Optimality: no, cannot be sure the first goal is the optimal one

2.4 Depth-limited Search Algorithm

The **depth-limited search algorithm** offers a solution to the completeness problem of the depth-first search algorithm by taking a parameter *deep* in input.
The procedure of the algorithm is the following:

1. Push the root node to the fringe
2. If the fringe is empty, then return failure
3. Remove the first node n from the fringe
4. If n is the goal, then return the solution
5. If n not_in(closed) and $d(n) < deep$, go ahead; otherwise repeat from step 2
6. Expand n
7. Add the children not in the explored list to the beginning of the fringe
8. Add n to the explored list
9. Repeat from step 2

We can only reach solutions that are at distance *deep* and we need knowledge to set a fixed *deep*.
The algorithm is evaluated as follows:

- Completeness: yes, if we know a bound to the solution depth
- Space Complexity: $O(b \cdot deep)$
- Time Complexity: $O(b^{deep})$
- Optimality: no

2.5 Iterative-deepening Search Algorithm

The **iterative-deepening search algorithm** is a depth-limited search where the depth is incremented step by step.
The procedure of the algorithm is the following:

1. Loop for depth $= 0$ to ∞
2. If Depth-Limited-Search(root, deep) == solution, then return the solution
3. Return failure

It is the best compromise between breadth-first search and depth-first search.
The algorithm is evaluated as follows:

- Completeness: yes, if we know a bound to the solution depth

- Space Complexity: $O(b \cdot d)$
- Time Complexity: $O(b^d)$
- Optimality: yes, if all costs are non-negative

2.6 Bidirectional Search

A problem related to search is its direction: in forward search, we start from the initial state and proceed to the goal, while in backward search, we start from the goal and go back to the initial state. The **bidirectional search algorithm** proceeds in both directions until the queues contain a common node.
The procedure of the algorithm is the following:

1. Initialize two queues, one for forward and one for backward search
2. Remove the first path from both the queues
3. Create new paths to the children
4. Reject a path if it contains a cycle
5. Add the new paths to the queues
6. Repeat from step 2 until the queues are not empty and do not share a node
7. If the queues do not share a node, then return the solution
8. Return Failure

The algorithm is evaluated as follows:

- Completeness: yes
- Space Complexity: $O(b^{d/2})$
- Time Complexity: $O(b^{d/2})$
- Optimality: yes

3 Heuristic Search Algorithms

Exhaustive blind search is not applicable for problems with exponential complexity. **Heuristic search algorithms** use problem specific strategies with some knowledge and experience to identify the most promising paths. These algorithms do not guarantee completeness and optimality, but find a good solution in a short time.

3.1 Heuristic Evaluation

The **heuristic evaluation** estimates the distance from a state to the goal:

$$h : s \to R \quad \text{such that} \quad h(goal) = 0$$

3.1.1 Admissible Heuristic

In **admissible heuristic**, the cost estimated to reach the goal is not higher than the lowest possible cost from the current node. Given a node n, a cost $h(n)$ from n to the goal, an optimal cost $h*(n)$ from n to the goal, then we have the following relation:

$$\forall n \mid h(n) \leq h*(n)$$

3.1.2 Monotonic Heuristic

In **monotone heuristic**, for each node n and its successor n', the estimated cost of reaching the goal from a node n is no longer greater than the step cost of getting its successor n' plus the estimated cost of reaching the goal from n'. Let $f(n)$ be the cost from the root to node n plus the cost estimation from n to the goal, then we have the following relation:

$$f(n) \leq f(n')$$

All monotonic heuristics guarantee local optimality and are admissible, but not all admissible heuristics are monotonic.

3.2 Best-first Search Algorithm

The **best-first search algorithm** evaluates each node estimating its distance to the goal, then sorts the fringe according to the estimated distances: the node with the best heuristic value is expanded.
The procedure of the algorithm is the following:

1. Push the root node to the fringe
2. If the fringe is empty, then return failure
3. Remove the first node n from the fringe
4. If n is the goal, then return the solution
5. If n is in the explored list, then repeat from step 2
6. Expand n
7. Add the children not in the explored list to the fringe
8. Add n to the explored list
9. Sort the fringe based on the heuristic values
10. Repeat from step 2

All nodes on the border are kept in memory. An alternative is represented by the beam-search algorithm that keeps in memory only the k states with the best evaluation, where k is the search radius.
The algorithm is evaluated as follows:

- Completeness: no
- Space Complexity: $O(b^m)$
- Time Complexity: $O(b^m)$
- Optimality: no

3.3 A* Algorithm

A algorithms are best-first search algorithms with an evaluation function:

$$f(n) = g(n) + h(n), \text{ where } h(n) \geq 0 \text{ and } h(goal) = 0$$

There exist some particular cases:

- if $f(n) = g(n)$, then it performs a uniform search
- if $f(n) = g(n) = depth(n)$, then it performs a breadth-first search
- if $f(n) = h(n)$, then it performs a greedy best first search

A* algorithm is an A algorithm with an admissible heuristic function:

$$f*(n) = g*(n) + h*(n), \text{ where:}$$

- $f*(n)$ = estimated cost of the path from the root to the goal
- $g*(n)$ = cost of the path from the root to node n
- $h*(n)$ = estimated cost of the path from node n to the goal

The algorithm expands the node with the smallest $f*(n)$. All nodes are kept in memory.
The algorithm is evaluated as follows:

- Completeness: yes, unless there are infinite nodes with $f \leq f(G)$
- Space Complexity: $O(b^m)$
- Time Complexity: $O(b^m)$
- Optimality: given a goal p
  1. Assume that a path p' is the shortest path to the goal
  2. Consider that a partial path p'' of p' is on the fringe
  3. Since p is expanded before p'', then we have:
     $$f(p) \leq f(p'')$$
  4. Since p is a goal, then:
     $$g(p) \leq g(p'') + h(p'')$$
  5. Since h is admissible, then for any path p' that extends p'':
     $$g(p'') + h(p'') \leq g(p')$$
  6. Finally, for any path p' to a goal:
     $$g(p) \leq g(p'),$$
     which contradicts point 1.

4 Constructive Greedy Algorithms

Constructive greedy algorithms take a data set and fastly produce a feasible solution through the following procedure:

1. Start from a random node n
2. Expand n
3. Choose the next node according to a local strategy
4. Extend the solution with the new node, which becomes the new n
5. Repeat from step 2 until a feasible solution is computed

4.1 Nearest Neighbors Algorithm

Nearest neighbors algorithm is one of the most common algorithms used to solve TSP problems.
Given n cities, the algorithm performs the following procedure,

1. Consider a starting tour consisting of a random city c_i
2. Add the closest new city c_{i+1} to the tour
3. Repeat step 2 until no new cities are available

The algorithm is not very efficient: the first edges are short while the final ones are long and the length of the tour with respect to the optimal tour length grows following a $log(n)$ formula.
The algorithm is evaluated as follows:

- Completeness: no
- Space Complexity: $O(n^2)$
- Time Complexity: $O(n^2)$
- Optimality: no, cannot guarantee optimality

4.2 Multi Fragment Algorithm

Multi fragment algorithm starts with the shortest edge and add the others in increasing order only if they do not create a $3-$degree city.
The procedure of the algorithm is the following:

1. Sort the edge costs in ascending order
2. Consider a tour with the the first edge
3. Add a new edge in incremental order only if it does not create a cycle
4. Repeat step 3 until no edge is available

The algorithm is evaluated as follows:

- Space Complexity: $O(n^2 log(n))$
- Time Complexity: $O(n^2 log(n))$
- Optimality: no, cannot guarantee optimality

4.3 Nearest Insertion Algorithm

Given n cities, the **nearest insertion algorithm** performs the following procedure:

1. Build an initial tour formed by the nearest cities
2. Choose a new node such that the Euclidean distance between it and two other nodes in the tour is minimum
3. Insert the new node between the two nodes of the tour
4. Repeat from step 2 until no new nodes are available

The algorithm is evaluated as follows:

- Space Complexity: $O(n^2)$
- Time Complexity: $O(n^2)$
- Optimality: no, cannot guarantee optimality

4.4 Farthest Insertion Algorithm

Given n cities, the **farthest insertion algorithm** performs the following procedure:

1. Build an initial tour formed by the farthest cities
2. Choose a new node such that the Euclidean distance between it and two other nodes in the tour is maximum
3. Insert the new node between the two nodes of the tour
4. Repeat from step 2 until no new nodes are available

The algorithm is evaluated as follows:

- Space Complexity: $O(n^2)$
- Time Complexity: $O(n^2)$
- Optimality: no, cannot guarantee optimality

5 Local Search Algorithms

Local search algorithms optimize solutions through the following procedure:

1. Start from a complete solution
2. Compute the best edge exchange that improves the solution
3. Execute the edge exchange
4. Repeat from step 2 until no improvement is possible

5.1 Hill Climbing Algorithm

Given a set of solutions A and the following objective function f:

$$\{f(s) \mid s \in A\},$$

then, we define a neighborhood function N that computes a subset of solutions $N(s)$ for each solution $s \in A$:

$$A \;\rightarrow\; 2^A,$$

The **hill climbing algorithm** explores the neighborhood searching for a better solution.
We can perform two different types of search: a greedy search and a simplified version. In the greedy search, if the best solution in $N(\mathit{current})$ is better than the actual best, then we restart from current, otherwise we stop. In the simplified version, as soon as we find a better neighborhood we continue the search from the associated solution, avoiding to visit all the neighborhoods.
The procedure of the algorithm is the following:

```
function Hill-Climbing(s, f, N):
    current = s
    while terminating condition is met:
        next = best solution in N(current)
        if f(next) < f(current):
            current = next
    return current
```

5.2 2-opt Algorithm

The **2-opt algorithm** removes crossings in the solution path by reversing the path between two nodes. The implementation of the **2-opt algorithm** is the following:

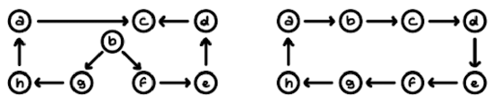
```
function 2-Opt():
    while best_gain != 0:
        best_gain = 0
        for i = 1 to n:
            for j = 1 to n:
                gain = Compute-Gain(i,j)
                if (gain < best_gain):
                    best_gain = gain
                    best_i = i
                    best_j = j
                    if first_improvement == true:
                        break
            if (best_gain < 0 && first_improvement == true):
                break
        Exchange(best_i, best_j)
```

The algorithm is evaluated as follows:

- Space Complexity: $O(n^2)$
- Time Complexity: $O(n^2)$

5.3 2.5-opt Algorithm

The **2.5-opt algorithm** performs the 2-opt algorithm procedure and check for two possible node shifts. A shift consists of moving a node to another position in the tour.

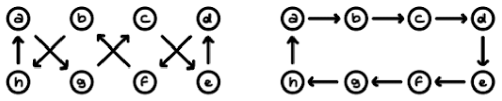


The algorithm is evaluated as follows:

- Space Complexity: $O(n^2)$
- Time Complexity: $O(n^2)$

5.4 3-opt Algorithm

The **3-opt algorithm** removes three links form the tour, obtaining three segments with which we check for 2-opt swaps and node shifts.



The algorithm is evaluated as follows:

- Space Complexity: $O(n^3)$
- Time Complexity: $O(n^3)$
- Optimality: TODO

6 Meta-heuristic Search Algorithms

Local search algorithms cannot escape from a local minimum and they are inefficient with large neighborhoods, thus either we accept solutions that are worst than the previous, or stochastically explore only a subset of a neighborhood through **meta-heuristic search algorithms**.
Meta-heuristic search algorithms have the following implementation:

```
function Meta-Heuristic-Search(s, f, N):
    current = s
    while terminating condition is met:
        stochastically compute next in N(current)
        Decide whether to continue with current = next
    return current
```

6.1 Simulated Annealing Algorithm

The **simulated annealing algorithm** is inspired by the natural annealing of metals and has the following procedure:

1. Start from an initial solution
2. Choose a random new solution from the neighborhood of the previous
    - If $f(next) < f(current)$, then $f(next)$ becomes the current node
    - Otherwise, given a temperature T, we use the following probabilistic function:

      $$\Delta E = f(next) - f(current)$$
      $$e^{-\frac{\Delta E}{T}}$$
3. Repeat from step 2 until the solution converges

Given a value $\beta = [1, 100]$, then the steps L from one temperature to the next are computed as follows:

$$L = \beta \cdot NN_length$$

Given the number of nodes n, then the temperature T is initialized as follows:

$$T = \frac{L}{\sqrt{n}}$$

Given a value $\alpha \approx 0.95$, then T decreases during the search as follows:

$$T_{i+1} = T \cdot \alpha$$

7 Genetic Algorithms

Genetic algorithms are based on Darwin's evolution theory: individuals are described by chromosomes, which are sets of genes; populations are sets of individuals; generations are sets of populations.
Individuals are evaluated using a **fitness function** representing their adaptation to the environment.

Genetic algorithms have the following implementation:

```
function Genetic-Algorithms():
    t = 0
    initialize P(t) with m individuals
    evaluate P(t)
    while terminating condition is not met:
        select parents from P(t)
        generate individuals with reproduction rules
        some individuals die in P(t)
        form population P(t + 1)
        evaluate population P(t + 1)
        t = t + 1
    return best individual
```

7.1 Ant Colony Algorithm

The basic principle of the **ant colony algorithm** is **stigmergy**, meaning a particular kind of indirect communication through which ants find the shortest paths in dynamic environments:

- While moving between nest and food, ants mark their path by pheromone laying
- Step-by-step decisions are biased by stigmergy

Pheromone is the colony's collective and distributed memory: it encodes the quality of local routing choices towards the destination target.
The implementation of the algorithm is the following:

```
function Ant-Colony():
    while terminating condition is not met:
        randomly position m ants on n cities
        for city = 1 to n:
            for ant = 1 to m:
                select next city through exploration and exploitation
                apply local trail updating rule
            compute length l of tour generated by m
        apply global trail updating rule using best ant
```