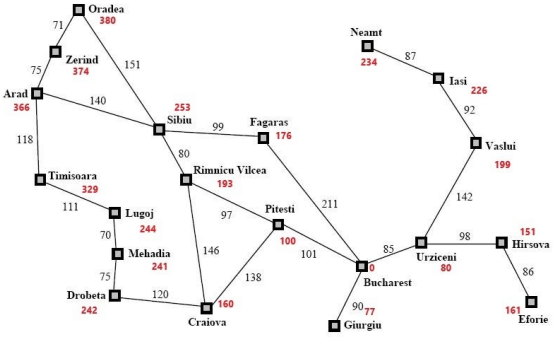
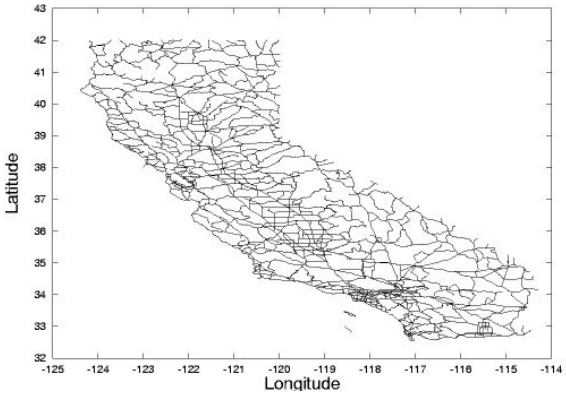
Algoritmi e strutture dati

Corso di Laurea in Ingegneria Informatica ed Elettronica – A.A. 2020-2021

DIPARTIMENTO DI INGEGNERIA

A* e algoritmi di ricerca su grafi





Marchetti Edoardo

Matricola 312879

edoardo.marchetti1@studenti.unipg.it

Sommario

1. **Descrizione del problema**

L'obiettivo dell'elaborato è la realizzazione di un algoritmo che permetta di calcolare in maniera efficace il cammino a costo minimo tra due vertici di un grafo pesato non orientato. Un esempio reale del problema può essere il calcolo del miglior percorso che permetta di raggiungere una città G a partire da una città S.

Il tema della ricerca del cammino minimo tra due punti può essere affrontato con varie tecniche. Supponiamo di avere un grafo come quello in figura 1, dove il nodo S è la città da cui partiamo e G invece è dove vogliamo arrivare.

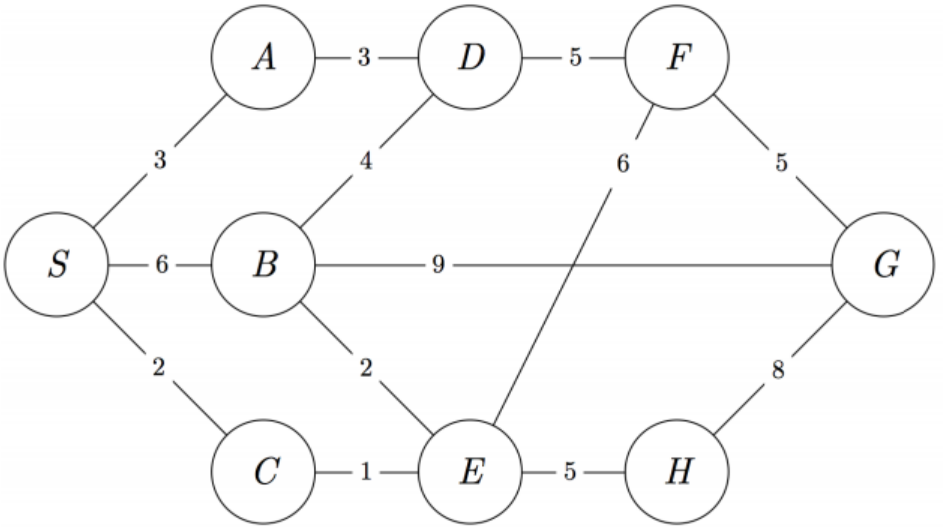


Figura 1. Grafo di esempio

1.1 **Possibili soluzioni**

Per trovare il percorso a costo minimo tra S e G si potrebbe pensare ad esempio di utilizzare una DFS. La soluzione finale sarebbe S -> A -> D -> F -> G. Come si può facilmente osservare il cammino indicato non è né il più corto in termini di nodi attraversati né il più "economico" dal punto di vista dei pesi.
Una seconda soluzione potrebbe essere visitare l'albero tramite BFS. In questo caso la soluzione trovata sarebbe S -> B -> G, la quale è già un miglioramento poiché ci permette di ottenere il percorso più breve se si considera come parametro il numero di nodi attraversati, ma non è ancora ciò che il problema richiede, ovvero il percorso a **costo minimo**.

Un primo miglioramento può essere quello di implementare un algoritmo definito nell'ambito degli agenti di ricerca come *Uniform-cost search* (UCS), il quale sviluppa la visita del grafo basandosi sul costo degli archi. In particolare UCS sfrutta una coda di priorità minima (min-heap) usando come valore chiave quello che d'ora in poi indicheremo come g(n), ovvero quanto si deve "spendere" per arrivare in n partendo

da S. In questo modo la soluzione che si ottiene è S -> C -> E -> B -> G. Il cammino indicato effettivamente è quello che permettere di raggiungere la destinazione indicata percorrendo il cammino a costo minimo. Quali sono però gli svantaggi di questa tecnica e come la si può migliorare? Il difetto principale di questo algoritmo, come in tutti gli algoritmi di ricerca non informata, è quello di esplorare il grafo in tutte le direzioni.

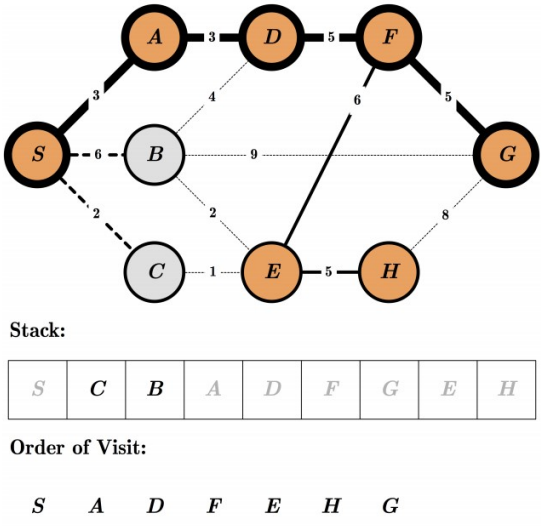


*Figura 2. Esplorazione del grafo tramite **DFS***

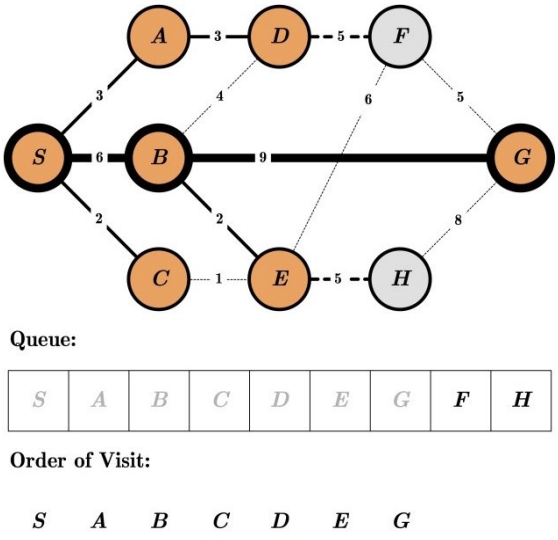


*Figura 3. Esplorazione del grafo tramite **BFS***

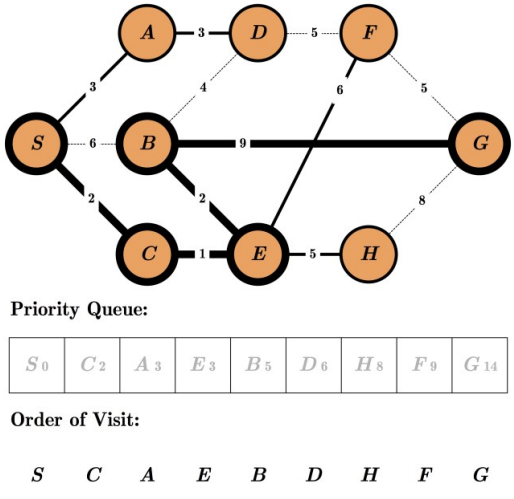


*Figura 4. Esplorazione del grafo tramite **UCS***

1.2 **Ricerca informata e A***

Le soluzioni proposte dunque o non trovano il percorso a costo minimo oppure non sono molto efficienti in quanto visitano il grafo in tutte le direzioni. Nel momento in cui si hanno delle informazioni su quello che verrà definito come **goal** (per tornare all'esempio delle città, la destinazione) si può pensare di aiutare l'algoritmo facendogli visitare il grafo in modo che i nodi visitati gli permettano di avvicinarsi alla soluzione.

Sempre nell'ambito degli agenti di ricerca gli algoritmi che operano in tali contesti prendono il nome di algoritmi di ricerca informata. L'informazione si può passare all'algoritmo nel caso della ricerca della strada più breve dalla città S alla città G è quella della distanza in linea d'aria tra un nodo n e la destinazione. Ciò prende il nome di **euristica** verrà indicata con h(n).

L'evoluzione di UCS diventa dunque un algoritmo anch'esso goloso, che denominiamo per chiarezza come *Greedy search*, ma che ordina i nodi da esplorare in base al valore dell'euristica e non più in funzione di g(n). Utilizzando Greedy search però non sempre si trova la soluzione ottima, ovvero il percorso a costo minimo.

Per sfruttare le potenzialità sia di UCS, il quale trova la strada a costo minimo, sia di Greedy search, capacità di visitare una porzione più ristretta del grafo, è necessario utilizzare l'algoritmo **A***. A* infatti prende come parametro di riferimento la somma dei due, ovvero f(n) = h(n) + g(n). In questo modo durante l'esplorazione del grafico si andrà a considerare sia quanto costa raggiungere il nodo n a partire dalla radice, sia quanto costerà al più raggiungere la destinazione a partire dal nodo n.

1.3 **L'importanza dell'euristica**

Nell'applicazione di A* è fondamentale la scelta dell'euristica. Infatti qualora l'euristica è ammissibile l'algoritmo restituirà sempre la soluzione migliore.
Un'euristica si dice ammissibile se non sovrastima mai il costo per raggiungere il goal, ovvero se per ogni nodo n, h(n) minore o uguale a h(n)*, dove quest'ultima indica il costo effettivo per raggiungere la destinazione a partire dal nodo n. Inoltre, nell'applicazione di A* su dei grafi, è necessario anche che l'euristica sia detta consistente, ovvero che per ogni n e per ogni suo successore n', h(n) deve essere minore o uguale a h(n') + w(n,n'), dove w(n,n') è il peso dell'arco che collega i due nodi.
L'ultima considerazione da fare sull'euristica è quella del compromesso tra complessità di calcolo e la qualità. Infatti un euristica che si avvicina al valore h(n)* permette all'algoritmo di trovare il percorso più velocemente, ma contemporaneamente sarà maggiore il tempo impiegato a calcolare il valore h(n). Dunque è necessario trovare un'euristica che bilanci i due aspetti.
Nell'ambito della ricerca della strada tra due città prendere come euristica la distanza in linea d'aria tra le due assicura entrambe le definizioni.

2. **Implementazione**

Per l'implementazione dell'algoritmo di base si è preso spunto dagli pseudo-codice riportato in figura 5 apportando poi alcune modifiche.

```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = g(n) + h(n)  */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

*Figura 5. Pseudo-codice A**

Come si può vedere la procedura prende come input due parametri che sono lo stato iniziale e quello finale, che nell'esempio a cui facciamo riferimento sono la città di partenza e di arrivo. Come prima cosa vengono creati due insiemi: **frontier** e **explored.** In particolare possiamo osservare come frontier è uno Heap, ovvero una coda di priorità, che viene inizializzato con lo stato di partenza. Questo insieme permette di avere a disposizione la lista dei nodi che possiamo visitare all'iterazione successiva del ciclo while in ordine crescente rispetto alla funzione f(n). L'insieme explored invece sarà composto da tutti quei nodi che l'algoritmo ha già visitato, così da evitare che la procedura entri in un loop infinito.
Successivamente si avvia un ciclo while fino a quando la lista frontier non è vuota (a quel punto significa che non ci saranno più nodi da visitare). A inizio iterazione si estrae dal min heap il nodo con la chiave minore e lo si aggiunge ad explored.
A questo punto si verifica se il nodo estratto è lo stato finale e se ciò non è vero si effettua l'operazione che prendi il nome di **expand**: si prende ogni nodo adiacente rispetto al nodo estratto da frontier e si effettuano due check. Nel caso in cui il neighbor non si trova nè in frontier nè in explored allora viene inserito nello heap, se invece si trova già nel frontier allora si decrementa la chiave, ovvero f(n).

Rispetto allo pseudo-codice riportato sopra nell'implementazione sono state apportate alcune modifiche che verranno spiegate nel paragrafo *2.3 Algorithms*.

2.1 **Classi di supporto**

Per una migliore comprensione del codice implementato vengono di seguito introdotte alcune classi di supporto.

Per rappresentare gli elementi **nodo** e **arco** che compongono un grafo sono state realizzate due classi *Node* e *Edge*.
Un oggetto Node ha come attributi un intero identificativo, le coordinate x e y sul piano, un valore handle che permette di gestire la posizione all'interno della lista di priorità, i campi g e h, una stringa che indica il colore del node e infine il campo parent.
L'oggetto Edge invece è costituito semplicemente da id, id del nodo di inizio e id del nodo di fine.

```
class Node:

    def __init__(self, id, x, y, g = math.inf, h= math.inf):
        self.id = id
        self.x = x
        self.y = y

        self.handle = -1

        self.g = g
        self.h = h

        self.color = 'white'

        self.parent = None
```

Figura 6. Costruttore Node

```
class Edge:

    def __init__(self, start, end, weight):
        self.startNode = start
        self.endNode = end
        self.weight = weight
```

Figura 7. Costruttore Edge

Per la costruzione del **grafo** si è invece pensato di realizzare una classe *Graph*, la quale tramite il suo costruttore permette di andare ad analizzare le liste di adiacenza di ogni singolo nodo. Come valori di input vengono prese la lista dei nodi e la lista dei vertici.

```
class Graph:

    def __init__(self, nodesMatrix, edgesMatrix):
        self.nodes = nodesMatrix
        self.adj_list = {}

        #Iniziazlizzo le liste per ogni nodo
        for node in self.nodes:
            self.adj_list[node.getID()] = []

        for edge in edgesMatrix:
            self.add_edge(edge)
```

Figura 8. Costruttore Graph

Successivamente è stata implementata la classe *Problem* la quale presenta i campi graph (rappresentato da un oggetto Graph), initialState che indica il nodo di partenza della ricerca e goalState, il quale invece indica il nodo di destinazione.

```
class Problem:

    def __init__(self, graph, initialState, goalState):
        self.graph = graph
        self.initialState = graph.getNode(initialState)
        self.goalState = graph.getNode(goalState)

        self.initialState.g = 0
```

Figura 9. Costruttore Problem

Come si può notare dalla figura 9, il costruttore della classe Problem pone a zero il campo g del nodo iniziale.

2.2 **Coda di priorità**

Per la risoluzione del problema, come è stato detto anche nei paragrafi di introduzione degli algoritmi, è necessario gestire la visita del grafo in base a delle funzioni di costo.
A questo scopo è stata implementata la classe *MinHeap,* il cui costruttore è illustrato nella figura 10. Si è deciso di inserire un nodo fittizio all'indice 0 così che gli indici validi per lo heap sono compresi tra 1 e heap_size.

```
class MinHeap:

    def __init__(self, key):
        self.heap = [Node(-1, -1, -1)]
        self.heap_size = 0

        if(key == 'g' or key == 'h' or key == 'h+g'):
            self.key = key

        else:
            raise Exception("Funzione di costo non valida")
```

Figura 10. Costruttore MinHeap

Le procedure che la compongono sono esattamente quelle viste a lezione:

- ***parent(i)***, ***left(i)***, ***right(i)***, le quali permettono di calcolare l'indice del nodo genitore, figlio sinistro e figlio destro del nodo in posizione i.
- ***is_empty()***, restituisce True se la dimensione dello heap è nulla.
- ***min_heapify (i)***, fa sì che dati due min-heap radicati in left(i) e right(i), l'albero radicato in i diventi un min-heap.

- ***minimum()***, restituisce l'elemento a chiave minima presente nello heap senza rimuoverlo.
- ***insert(node)***, inserisce correttamente il nodo node all'interno dello heap.
- ***decrease_key(node, k)***, permette di impostare k come chiave di node qualora k sia minore rispetto al valore chiave attuale.
- ***delete(node)***, permette di eliminare l'elemento node dallo heap.
- ***extract_min(),*** restituisce l'elemento a chiave minima presente nello heap e lo rimuove.

Come già ampiamente discusso in classe, ad eccezione di minimum e delle tre funzioni di utilità per il calcolo dell'indice, tutte le procedure hanno una complessità logaritmica.

2.3 **Algorithms**

Ora che tutte le classe di supporto sono state introdotte si procede alla spiegazione dell'implementazione dell'algoritmo spiegando le differenze rispetto allo pseudo-codice della figura 5 e perché sono state introdotte.

Come si può subito notare dalla figura 11, la procedura non prende il nome di A*, ma bensì di ***best_first***. Questo perchè lo stesso codice può essere utilizzato per risolvere il problema tramite UCS, Greedy_search o A* in funzione del valore che viene assegnato al parametro di input *f*. Nel file *Algorithms* sono presenti anche tre metodi chiamati con il nome dei tre algoritmi che permettono di invocare direttamente best_first con la funzione di costo rispetto alla quale dovranno essere riordinati i nodi all'interno dello heap. Le procedure ***greedy_search*** e ***a_start***, inoltre, invocano anche la funzione *computeHeuristic()* sull'oggetto problem per poter impostare i valori dei parametri h in ogni nodo del grafo rispetto al goalState.

La funzione best_first restituisce come risultato una lista ottenuta tramite la procedura solution invocata su node, il costo totale del percorso e una seconda lista che comprende tutti i nodi esplorati durante la ricerca.

All'interno di *best_first* prima di tutto si verifica se l'initialState coincide con il goalState del problema. In quel caso non sarà necessario continuare e di conseguenza vengono restituiti una lista che comprende il solo initialState, un costo nullo per il percorso e una lista di nodi esplorati vuota.

Successivamente vengono inizializzati gli insieme **frontier** e **explored**: il primo è un MinHeap al quale viene indicato tramite f rispetto a quale parametro dover riordinare i nodi, il secondo invece è una semplice lista. Come visto nello pseudo-codice precedente all'interno di frontier viene inserito il nodo di partenza.

```python
def best_first(problem, f = 'g'):

    if problem.goalState == problem.initialState:
        return (problem.initialState.solution(), 0, [])

    frontier = MinHeap(f)
    frontier.insert(problem.initialState)
    explored = set()

    while not frontier.is_empty():
        node = frontier.extract_min()

        if problem.goalState == node:
            return (node.solution(), node.g, explored)

        explored.add(node.getID())
        node.color = 'black'

        for [neighbor, edgeCost] in node.expand(problem):

            if neighbor.color == 'white':
                #Aggiungo il nodo a frontier
                neighbor.color = 'gray'
                neighbor.parent = node
                neighbor.g = neighbor.parent.g + edgeCost

                frontier.insert(neighbor)

            elif neighbor.color == 'gray':
                #Nodo in frontier
                if f != 'h' and neighbor.g > node.g + edgeCost:
                    neighbor.parent = node
                    neighbor.g = node.g + edgeCost
                    frontier.decrease_key(neighbor, neighbor.g)

    return (None, math.inf, explored)
```

Figura 11. Codice di best_first

Completata questa prima fase si entra nel ciclo while il quale continuerà fino a quando la coda di priorità non si svuota. Se non ci sono più nodi da esplorare allora l'algoritmo restituisce un None per indicare che non esiste la soluzione, un costo infinito poichè il nodo di destinazione non è raggiungibile e la lista dei nodi esplorati durante tutte le iterazioni.

Ad ogni iterazione del while viene estratto il nodo a chiave minima per poi verificare se questo è quello di destinazione. Se sì allora la procedura restituisce la lista prodotta da *solution()* invocata sullo stesso nodo, il valore del campo g e la lista dei nodi esplorati fino a quel momento. Nel caso in cui il minimo dello heap non corrisponde al goalState allora si "colora" il nodo di nero e lo si inserisce in explored.

Fatto ciò si passa alla fase di *expand*. La procedura *expand(problem)* invocata sul node permette di ottenere la liste di coppie (neighbor, edgeCost).

```
def expand(self, problem):
    graph = problem.graph

    adjacent = []

    for (adjId, weight) in graph.adj_list[self.id]:
        adjacent.append((graph.getNode(adjId), weight))

    return adjacent
```

Figura 12. Procedura exapnd della classe Node

Per ogni nodo adiacente a questo punto si fa un check rispetto al colore del nodo. Rispetto allo pseudo-codice si è preferito sfruttare il campo color del nodo per non dover andare ad effettuare una ricerca che comporterebbe un considerevole aumento in termini di numero di operazioni da dover eseguire che può portare ad un calo delle prestazioni qualora il numero di nodi presenti in frontier e explored sia elevato.
Nel caso in cui il colore del **neighbor** sia bianco allora significa che il nodo non è inserito in nessuna delle due liste. Per questo motivo si imposta il colore a gray, si aggiorna il campo parent e infine si setta il valore di g, per poi inserire il nodo nella coda di priorità.
Se invece il colore del nodo è gray allora il nodo si trova in frontier e non in explored. Dunque si verifica se è stato individuato un percorso migliore che collega l'initialState al neighbor (nel caso in cui si usi come f g(n) o h(n)+g(n)). Se ciò si verifica allora si aggiorna il genitore del nieghbor, il valore di g e si invoca decrease_key(neighbor, neighbor.g) sul frontier così da riposizionare correttamente il nodo all'interno della coda di priorità.

2.4 **Considerazioni sulle scelte implementative**

Durante l'implementazione dell'algoritmo sono state affrontate due decisioni su tutte: quale coda di priorità utilizzare e in che forma rappresentare il grafico.

Le motivazioni che hanno portate ad utilizzare un min-heap, piuttosto che uno heap binomiale ad esempio, è stato il fatto di non avere necessità di dover utilizzare una procedura di unione tra due heap, la quale per gli heap binari ha una complessità lineare.

Per quanto riguardo il grafo invece la scelta è ricaduta sull'utilizzo della lista di adiacenza in quanto si è notato che la maggior parte dei nodi hanno un numero relativamente basso di vicini rispetto al numero complessivo degli elementi che compongono il grafo. In questo modo dunque si evita di sprecare spazio in memoria. Un'altra considerazione fatta è che all'interno di UCS, Greedy Search e A* non è necessario verificare se due nodi A e B sono adiacenti, ma nella fase di expand si richiedono semplicemente i vicini di un nodo.

3. **Analisi sulla complessità**

Per effettuare un corretto studio della complessità del codice prodotto si preferisce mostrare anche la complessità dei passaggi chiave effettuati anche dalle classi di supporto illustrate nel precedente capitolo.

La costruzione del grafo, come mostrato in figura 8, comprende due cicli for: il primo di complessità O(n), dove n indica il numero di nodi, e il secondo invece O(m), dove m implica il numero di archi. Nei file utilizzati come esempio, presi da https://www.cs.utah.edu/~lifeifei/SpatialDataset.htm, si è notato che il numero di archi è molto simile al numero di vertici e quindi si può assumere che la complessità sia in termini temporali che di memoria valga O(n).

Le procedure presenti nella struttura dati MinHeap come accennato già in precedenza hanno tutte un costo logaritmico ad esclusione dell'inizializzazione e delle procedure parent(i), left(i), right(i) e is_empty().

Per quanto riguarda la complessità della procedura best_first si deve tener conto di quale funzione di costo è utilizzata per l'ordinamento dei nodi. Infatti come indicato nel primo capitolo lo scopo di utilizzare Greedy search o A* piuttosto che UCS ha lo scopo di far raggiungere all'algoritmo la destinazione il più velocemente possibile visitando un minor numero di nodi. Un minor numero di nodi visitati infatti implica una quantità di iterazioni del ciclo while più piccolo prima di trovare il percorso a costo minimo verso la destinazione se esiste.

Stando al libro *Artificial Intelligence: A modern Approach* le complessità temporali e spaziali dei tre algoritmi sono le seguenti:

- $O(b^{1+C^*/\varepsilon})$ per UCS, dove b è il fattore di branching, ovvero il grado medio dei nodi nell'albero, C* è il costo della soluzione ottima e ε è il costo minimo di un arco
- $O(b^m)$ per Greedy Search, dove m è il diametro del grafo
- $O(b^{\varepsilon d})$ per A*, dove d indica la lunghezza del cammino initialState-goalState in termini di nodi e ε indica l'errore relativo fatto dall'euristica che è pari a $(h^* - h)/h^*$ con h* è il costo reale del percorso che dalla partenza porta alla destinazione.

Nel caso di Greedy Search e A* l'uso di una euristica migliore permette di diminuire la complessità temporale delle procedure.

Analizzando il codice prodotto in termini di numero di nodi, n, e di vertici, m, che compongono il grafo si può dire che la complessità totale di best_first nel caso in cui tutti i nodi vengano visitati è O(m * lg n). Infatti si può ragionare in maniera simile a quanto fatto con Dijkstra a lezione:

- ogni nodo viene inserito ed estratto dal frontier al più una volta, dunque il massimo numero di iterazioni del ciclo while è n
- ad ogni iterazione del ciclo while si esegue un'operazione di extract_min (complessità totale O(n* lg n)) e un ciclo for che genera al più m iterazioni grazie alla rappresentazione con liste di adiacenza del grafo.
- ad ogni iterazione del ciclo for viene eseguita al più un'operazione di insert o di decrease_key, le quali hanno complessità O(lg n). Dunque il costo totale del ciclo for è O(m * lg n)
- Il costo totale sarà dunque O((n+m) * lg n) che diventa O (m lg n) se ogni vertice è raggiungibile dall'initialState.

Ultima considerazione da fare è il calcolo dell'euristica. Infatti le procedure *a_star* e *greedy_search*, prima di chiamare best_first, invocano il metodo *computeHeuristic()* sull'oggetto problem, il quale ha una complessità O(n).

4. **Esempi di applicazioni e discussione risultati**

Prima di eseguire l'algoritmo su i dati reali ricavati dal link indicato nel capitolo 3 ci si aspettava che:

- UCS trovi sempre la strada a costa minimo, ma visitando un numero maggiore di nodi .
- Greedy Search non trovi sempre la soluzione migliore, ma visiti il minor numero di nodi tra i tre
- A* restituisca sempre il percorso a costo minimo e visiti un numero di nodi inferiore a UCS, ma più elevato rispetto a Greedy Search.

(I dati sperimentali sono riportati all'interno del file "Dati applicazioni algoritmi" allegato alla relazione)

Come si può vedere dai dati riportati le tre ipotesi iniziali sono rispettate. Inoltre si può vedere come Greedy Search impieghi un tempo nettamente inferiore all'aumentare della dimensione del grafo rispetto a UCS e anche ad A*.

5. **Considerazioni finali**

In conclusione l'algoritmo A* risulta avere dei pregi e dei difetti. Infatti se da una parte permette di ottenere sempre il percorso a costo minimo, dall'altra il numero di nodi visitati porta ad avere un tempo di esecuzione e un'occupazione di memoria (rappresentato dal numero di nodi visitati) molto elevati rispetto a Greedy Search nel momento in cui lo spazio di ricerca diventa più ampio o semplicemente si allontanano i punti initialState e goalState. In particolare è questo secondo aspetto che a volte rende inutilizzabile l'algoritmo. Per questo motivo esistono delle versioni di A* che limitano l'uso di memoria come ad esempio IDA*, RBFS e SMA*.

6. **Risorse**

- Wikipedia : https://en.wikipedia.org/wiki/A*_search_algorithm
- https://www.cs.utah.edu/~lifeifei/SpatialDataset.htm
- "Artificial Intelligence: A Modern Approach", libro di Peter Norvig e Stuart J. Russell
- Dispense del corso "Artificial Intelligence (AI)" della Columbia University sulla piattaforma edX.