Tracce delle tesine per il corso di Algoritmi e Strutture Dati a.a. 2019-2020

1) Implementazione dell'algoritmo di ordinamento Natural Mergesort 2
2) Implementazione dell'algoritmo di ordinamento Shellsort 3
3) Algoritmo di Knuth-Morris-Pratt per il problema dell Pattern Matching 4
4) Realizzazione di una struttura dati trie per l'indicizzazione di testi 5
5) Implementazione di un algoritmo per il calcolo della distanza di editing tra due stringhe 7
6) Implementazione di un algoritmo per la stampa accurata di un testo 8
7) Implementazione dell'algoritmo per il calcolo del codice di Huffmann 9
8) Implementazione di un gestore della cache ottimo 10
9) Implementazione dell'algoritmo di Kruskal 11
10) Implementazione dell'algoritmo di Prim 12
11) Scrittura di un algoritmo per la generazione di labirinti 13
12) Implementazione dell'algoritmo Dijkstra 14
13) Implementazione dell'algoritmo A* 15
14) Implementazione di algoritmo per il calcolo dell'inviluppo convesso di un insieme di punti 16
15) Implementazione di un algoritmo sweep line per il calcolo di intersezioni tra segmenti nel piano 17

1) Implementazione dell'algoritmo di ordinamento Natural Mergesort

Natural mergesort è una variante di Mergesort che procede nel seguente modo: l'array viene prima di tutto suddiviso in sottosequenze ordinate. Tali sequenze vengono poi fuse (come nel Mergesort tradizionale) procedendo dal basso verso l'alto (cioè le sequenze ottenute dalla fusione delle sottosequenze originali, vengono a loro volta fuse, quelle ottenute in questo modo vengono ancora fuse, e così via). Il progetto consiste nell'implementare Natural Mergesort e nell'effettuare un'analisi sperimentale confrontando le prestazioni di Natural Mergesort con quelle degli altri algoritmi di Sorting. A tal fine è possibile utilizzare le classi viste nell'esercitazione di laboratorio sugli algoritmi di ordinamento.

2) Implementazione dell'algoritmo di ordinamento Shellsort

L'algoritmo di ordinamento Shellsort è una variante di Insertionsort che si basa sull'osservazione che Insertionsort è abbastanza veloce se eseguito su un array non troppo disordinato. L'idea è quindi quella di effettuare diverse passate sull'array. Ogni passata è una esecuzione di Insertionsort in cui si confrontano (ed eventualmente scambiano) elementi che sono ad una certa distanza **h**, con **h** che si riduce ad ogni passata fino a divenire 1 (valore per il quale quindi si esegue l'InsertionSort classico).

Le varie passate operano su sottoinsiemi sempre più grandi e quindi potenzialmente sono sempre più costose. Il maggior costo che si avrebbe nelle passate successive viene però controbilanciato dal fatto che ad ogni passata il livello di disordine dell'array diminuisce. L'ultima passata è un Insertionsort puro che però opera su una sequenza già quasi ordinata. Lo pseudocodice di Shellsort è mostrato di seguito.

Le prestazioni di Shellsort dipendono dalla sequenza di valori **h** che si utilizza. La sequenza usata è detta *gap sequence*. In letteratura sono state proposti diversi modi di generare gap sequence. La versione originale di Shellsort usa una gap seqeunce che partendo da $\frac{n}{2}$, procede dividendo sempre per due il valore precedente. Altre gap sequence sono, la sequenza di Pratt, di Hibbard, di Knuth, e di Segdwick.

Per ulteriori dettagli su shellsort vedere la seguente pagina https://it.wikipedia.org/wiki/Shell_sort.

Il progetto consiste nell'implementare Shellsort con diverse tipi di gap sequence e nell'effettuare un'analisi sperimentale confrontando le diverse versioni di Shellsort tra di loro e con gli altri algoritmi di Sorting. A tal fine è possibile utilizzare le classi viste nell'esercitazione di laboratorio sugli algoritmi di ordinamento.

3) Algoritmo di Knuth-Morris-Pratt per il problema dell Pattern Matching

Il problema del Pattern Matching consiste nel trovare la locazione di uno specifico *text pattern* in un documento testuale di grandi dimensioni. L'approccio brute force per il pattern matching confronta il pattern ed il testo un carattere alla volta finché non si trova una corrispondenza o il testo finisce. La complessità di tale algoritmo nel caso peggiore è $O(n \cdot m)$ essendo n la lunghezza del testo e m la lunghezza del pattern. Un algoritmo più efficiente è l'algoritmo di Knut Morris e Pratt.

L'algoritmo di Knuth-Morris-Pratt (KMP) cerca di eliminare la principale inefficienza dell'algoritmo brute-force: quando si testa un possibile piazzamento del pattern si effettua un certo numero di confronti ma non appena si ha un mismatch tutta l'informazione ricavata dai confronti precedenti viene persa. L'idea dell'algoritmo KMP è quella di riutilizzare, per quanto possibile, l'informazione ricavata dai passi precedenti.

Per ulteriori dettagli sull'algoritmo KMP si veda il materiale aggiuntivo.

Il progetto consiste nell'implementare l'algoritmo KMP e nell'effettuare un'analisi sperimentale confrontandolo con l'algoritmo brute force su diversi testi e pattern reali. Nell'analisi sperimentale vanno misurati i tempi di calcolo e il numero di confronti effettuati dai due algoritmi.

4) Realizzazione di una struttura dati trie per l'indicizzazione di testi

Un trie è una struttura dati gerarchica per la memorizzazione di stringhe. I trie sono adatti per operazioni di Information Retrieval. Le operazioni principali sui Tries sono Pattern matching e Prefix Matching.

Sia S un insieme di s stringhe definite su un alfabeto Σ e tali che nessuna stringa sia un prefisso di un'altra. Uno *Standard Trie* per S è un albero T così definito (si veda anche l'esempio in figura):

- Ogni nodo di T, eccetto la radice, è etichettato con un carattere di Σ
- L'ordine dei figli di un nodo interno è determinato dall'ordinamento alfabetico di Σ
- T ha s nodi foglia, ognuno associato con una stringa di S, tale che la concatenazione delle etichette dei nodi nel cammino dalla radice a un nodo foglia v di T fornisce la stringa di S associata a v.

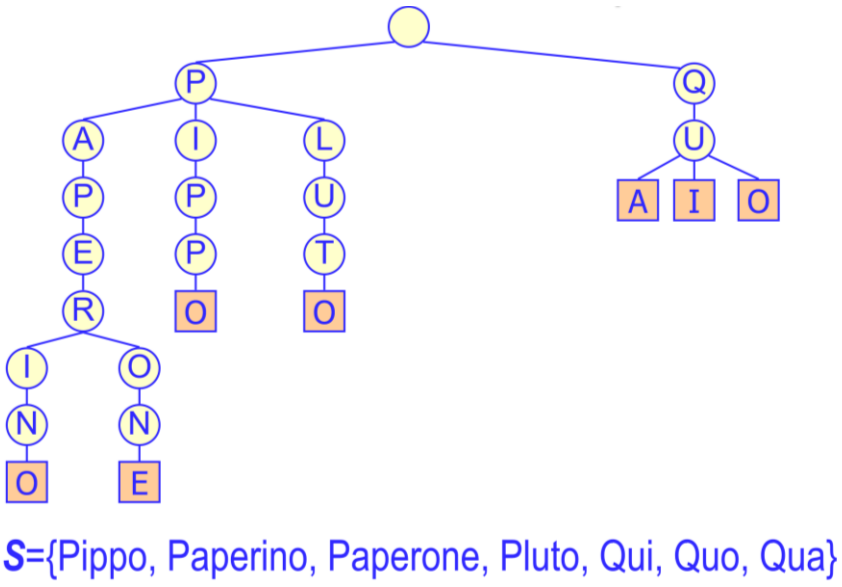


Figura 1 Un esempio di Standard trie

In uno Standard Trie ogni cammino dalla radice ad un nodo interno v di profondità i corrisponde a un prefisso di lunghezza i di una qualche stringa in S. Se più stringhe in S hanno un prefisso comune di lunghezza i, i cammini dalla radice ai nodi foglia che le rappresentano hanno un sottocammino comune dalla radice ad un nodo a profondità i. I tries memorizzano efficientemente prefissi comuni tra diverse stringhe. Uno standard trie T può essere usato per implementare un dizionario le cui chiavi sono le stringhe in S. Volendo cercare una stringa $X \in S$ nel dizionario, si cerca, a partire dalla radice di T, il cammino indicato dai caratteri di X. Se tale cammino esiste e termina in un nodo foglia allora X è nel dizionario.

In uno standard trie, si ha una certa inefficienza nell'uso dello spazio. In particolare ci possono essere molti nodi con un solo figlio. Tale situazione porta ad avere un numero di nodi che è $O(n)$, dove n è la lunghezza totale delle stringhe. Il numero di nodi può quindi essere molto più grande del numero di stringhe rappresentate. Un *compressed trie* è una struttura dati che riduce lo spreco di spazio di uno standard trie. In un compressed trie non ci sono nodi con un solo figlio; catene di nodi con un solo figlio vengono compresse in un unico nodo. Un compressed trie è utile quando lo si usa come una struttura ausiliaria di indicizzazione di una collezione di stringhe memorizzate in una struttura primaria. In questo caso non memorizziamo nel trie tutti i caratteri delle stringhe, ma solo i loro indici. Per maggiori dettagli sui tries si veda il materiale aggiuntivo.

Il progetto consiste nell'implementare la struttura dati compressed trie. Dato un testo, si deve costruire il trie e permettere all'utente di cercare la posizione di una parola nel testo. Il programma va poi testato con vari testi reali misurando i tempi id creazione del trie e i tempi di interrogazione.

5) Implementazione di un algoritmo per il calcolo della distanza di editing tra due stringhe

La similarità tra due stringhe può essere misurata in vari modi. Uno di questi utilizza la *distanza di editing*, che è definita in base al numero e al tipo di operazioni di modifica necessarie per trasformare la prima stringa nella seconda. Si considerino le seguenti operazioni in cui x è la stringa di input, z è la stringa che viene costruita eseguendo le varie operazioni, mentre gli indici i e j indicano il carattere corrente in x e il carattere corrente in z, rispettivamente.

- **Copia** un carattere da x a z, impostando $z[j] = x[i]$ e poi incrementando i e j. Questa operazione esamina $x[i]$.
- **Sostituisci** un carattere di x con un altro carattere c, impostando $z[j] = c$ e poi incrementando i e j. Questa operazione esamina $x[i]$.
- **Cancella** un carattere di x, incrementando i, senza modificare j. Questa operazione esamina $x[i]$.
- **Inserisci** il carattere c in z, impostando $z[j] = c$ e poi incrementando j, senza modificare i. Questa operazione non esamina i caratteri di x.
- **Scambia** i prossimi due caratteri copiandoli da x a z, ma in ordine inverso; per fare questo, impostiamo prima $z[j] = x[i + 1]$ e $z[j + 1] = x[i]$ e, poi, $i = i + 2$ e $j = j + 2$. Questa operazione esamina $x[i]$ e $x[i + 1]$.
- **Distruggi** la parte restante di x, impostando $i = m + 1$. Questa operazione esamina tutti i caratteri di x che non sono stati ancora esaminati. Se questa operazione viene svolta, deve essere l'ultima.

Per esempio, un modo per trasformare la stringa di input `algorithm` nella stringa di output `altruistic` consiste nell'utilizzare questa sequenza di operazioni (i caratteri sottolineati sono $x[i]$ e $z[j]$ dopo ogni operazione):

Operazione	x	z
	algorithm	_
copia	algorithm	a_
copia	algorithm	al_
sostituisci con `t`	algorithm	alt_
cancella	algorithm	alt_
copia	algorithm	altr_
inserisci `u`	algorithm	altru_
inserisci `i`	algorithm	altrui_
inserisci `s`	algorithm	altruis_
scambia	algorithm	altruisti_
inserisci `c`	algorithm	altruistic_
distruggi	algorithm_	altruistic_

In realtà esistono molti altri modi per trasformare `algorithm` in `altruistic`. Se assumiamo che ogni operazione abbia un costo associato possiamo calcolare il costo di una sequenza di operazioni. Date due sequenze $x[1 \,..\, m]$ e $y[1 \,..\, n]$ e un insieme di costi per ogni operazione, la ***distanza di editing*** tra x e y è il costo della sequenza di operazioni più economica che trasforma x in y. Per maggiori dettagli si veda il materiale aggiuntivo.

Il progetto consiste nell'individuare ed implementare un algoritmo di programmazione dinamica che calcola la distanza di editing tra due stringhe date x e y e stampa una sequenza di operazioni ottima. L'algoritmo va poi testato su varie coppie di stringhe e per vari insiemi di costi misurando i tempi di esecuzione.

6) Implementazione di un algoritmo per la stampa accurata di un testo

Considerate il problema di stampare in modo accurato un paragrafo di testo con una stampante. Il testo di input è una sequenza di n parole di lunghezza $l_1, l_2, \ldots, l_n$. Vogliamo stampare questo paragrafo in modo accurato su un certo numero di righe, ciascuna delle quali contiene al massimo M caratteri. Il nostro criterio di "stampa accurata" è il seguente. Se una data riga contiene le parole da i a j, con $i \leq j$, e lasciamo esattamente uno spazio fra le parole, il numero di spazi extra alla fine della riga è $M - j + i - \sum_{k=i}^{j} l_k$, che deve essere non negativo, in modo che le parole possano adattarsi alla riga. Per ottenere una stampa accurata si vuole rendere minima la sommatoria, per tutte le righe tranne l'ultima, dei cubi dei numeri di spazi extra che restano alla fine di ogni riga. Per maggiori dettagli si veda il materiale aggiuntivo.

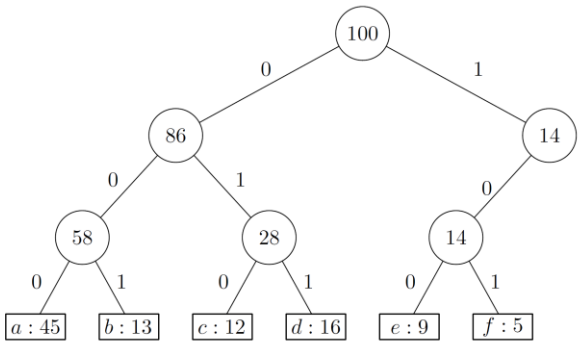
Il progetto consiste nell'individuare ed implementare un algoritmo di programmazione dinamica per la stampa in maniera accurata di un paragrafo di n parole. L'algoritmo va poi testato su vari esempi di testo misurando i tempi di esecuzione.

7) Implementazione dell'algoritmo per il calcolo del codice di Huffmann

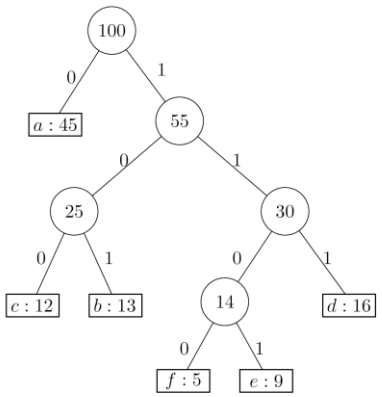
Il codice di Huffman è un codice di codifica molto diffuso per comprimere i dati testuali. Dato un testo da comprimere, si crea una tabella che indica la frequenza di ciascun carattere nel testo. L'idea è di usare pochi bit per i caratteri più frequenti e più bit per i caratteri meno frequenti. Il codice di Huffman, così come qualunque altro codice prefisso (cioè un codice in cui ciscun parola di codice non è prefisso di nessun'altra parola di codice), può essere rappresentato mediante un albero binario che può essere utilizzato nella fase di decodifica per individuare facilmente le parole di codice. L'albero binario associato ad un codice prefisso può essere definito come segue:

- ogni foglia dell'albero è un carattere;
- ogni arco padre-figlio è associato al valore 0 o 1 di un bit (figlio sinistro=0, figlio destro=1);
- la parola di codice associata ad un carattere è la sequenza di bit lungo il cammino dalla radice alla foglia.

Gli alberi binari associati a due codici prefissi (uno di lunghezza fissa e uno di lunghezza variabile) sono mostrati in Figura. Per maggiori dettagli si veda il capitolo 16 del libro di testo.



carattere	a	b	c	d	e	f
frequenza	45	13	12	16	9	5
codice	000	001	010	011	100	101



carattere	a	b	c	d	e	f
frequenza	45	13	12	16	9	5
codice	0	101	100	111	1101	1100

Il progetto consiste nel creare un programma che prende in input un testo, ne calcola la tabella delle frequenze, calcola il codice di Huffmann e permette di codificare e decodificare il testo. Il programma va poi testato su vari testi di prova, misurando i tempi di esecuzione delle varie fasi (creazione della tabella, calcolo del codice, codifica, decodifica) e il fattore di compressione.

8) Implementazione di un gestore della cache ottimo

I computer moderni utilizzano una cache per memorizzare una piccola quantità di dati in una memoria veloce. Anche se un programma può accedere a grandi quantità di dati, memorizzando un piccolo sottoinsieme di dati.della memoria principale nella cache - una piccola ma più veloce memoria - tempo di accesso globale può diminuire notevolmente. Quando un programma per computer viene eseguito, crea una sequenza $\langle r_1, r_2, \dots, r_n \rangle$ di n richieste di memoria, dove ogni richiesta riguarda un dato elemento. Ad esempio, un programma che accede a 4 elementi distinti $\{a, b, c, d\}$, potrebbe generare la sequenza di richieste $\langle d, b, b, d, d, b, d, a, c, c, d, b, a, c, c, b \rangle$. Sia k la dimensione della cache. Quando la cache contiene k elementi e il programma richiede il $(k+1)$-esimo elemento, il sistema deve decidere quali k elementi tenere nella cache. Più precisamente, per ogni richiesta r_i, l'algoritmo di gestione della cache controlla se l'elemento r_i è già nella cache. Se sì allora abbiamo un *cache hit*, altrimenti abbiamo un *cache miss*. Se si ha un cache miss, il sistema recupera la risorsa r_i dalla memoria principale, e il gestore della cache deve decidere se tenere r_i nella cache. Se decide di mantenere r_i e la cache contiene già k elementi, deve rimuovere un elemento per fare spazio. L'algoritmo di gestione della cache rimuove gli elementi dalla cache con l'obiettivo di ridurre al minimo il numero di cache miss sull'intera sequenza di richieste.

In genere, la cache è un problema on-line. Cioè, dobbiamo prendere decisioni su quali dati conservare nella cache senza conoscere le future richieste. Se si considera la versione off-line del problema, cioè quella in cui si conosce in anticipo l'intera sequenza di n richieste e la dimensione della cache k, è possibile determinare le rimozioni dalla cache in maniera da minimizzare il numero totale di cache miss utilizzando un algoritmo goloso che rimuove la risorsa che verrà richiesta più lontano nel futuro. Per maggiori dettagli riguardo tale algoritmo si veda il materiale aggiuntivo.

Scopo del progetto è implementare un gestore della cache che riceve una sequenza di richieste e determina la sequenza di rimozioni ottima. Tale gestore va poi valutato sperimentalmente su varie sequenze di test misurando i tempi di gestione di una richiesta e il numero di cache miss.

9) Implementazione dell'algoritmo di Kruskal

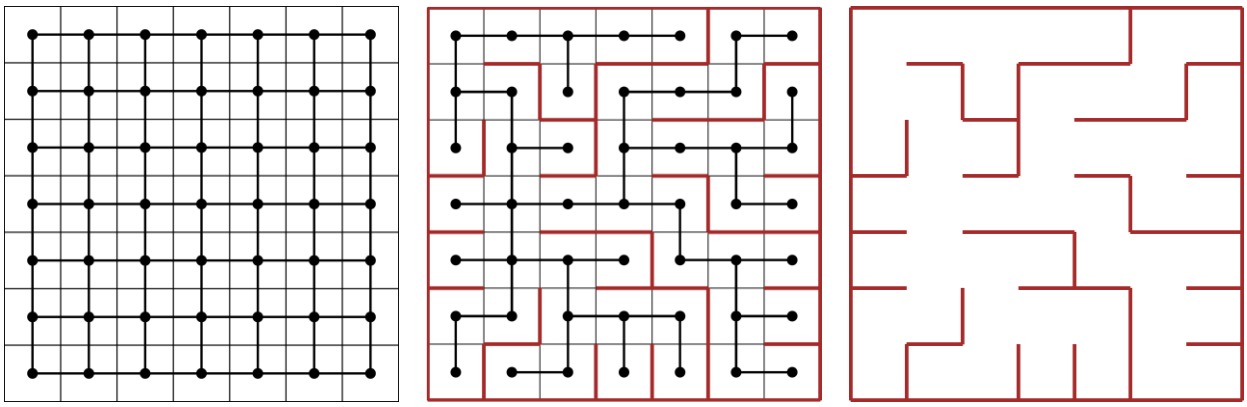
Il progetto consiste nell'implementare l'algoritmo di Kruskal per il calcolo di un Minimum Spanning Tree e nell'effettuare una valutazione sperimentale delle sue prestazioni per varie tipologie di grafi di input. L'attività sperimentale deve prevedere la generazione casuale di vari insiemi di grafi per diversi valori di n (numero di vertici) ed m (numero di archi). Più precisamente per ogni valore di densità (definita come il rapporto m/n) vanno generati un certo numero di grafi per valori crescenti di n. Per maggiori dettagli vedere i capitoli 23 (per l'algoritmo di Kruskal) e 21 (per l'implementazione della Union-find) del libro di testo.

10) Implementazione dell'algoritmo di Prim

Il progetto consiste nell'implementare l'algoritmo di Prim per il calcolo di un Minimum Spanning Tree e nell'effettuare una valutazione sperimentale delle sue prestazioni per varie tipologie di grafi di input. L'attività sperimentale deve prevedere la generazione casuale di vari insiemi di grafi per diversi valori di n (numero di vertici) ed m (numero di archi). Più precisamente per ogni valore di densità (definita come il rapporto m/n) vanno generati un certo numero di grafi per valori crescenti di n. Per maggiori dettagli vedere il capitolo 23 (per l'algoritmo di Prim) del libro di testo.

11) Scrittura di un algoritmo per la generazione di labirinti

Il progetto consiste nello scrivere un programma per la generazione di labirinti. Un labirinto può essere generato partendo da una disposizione di celle predeterminata (di solito una griglia rettangolare, ma altre disposizioni sono possibili). Tale disposizione di celle può essere vista come un grafo connesso G i cui nodi rappresentano le celle e i cui archi rappresentano potenziali passaggi tra celle adiacenti. Il labirinto può quindi essere generato individuando un sottografo G'di G in cui è difficile trovare un percorso tra due nodi s e t specificati. Se G' fosse non connesso, ci sarebbero regioni del labirinto non raggiungibili e quindi inutili; se ci fossero cicli potrebbero esistere cammini multipli tra s e t. Il problema di generare un algoritmo viene quindi di solito risolto individuando uno spanning tree casuale del grafo G. In figura è mostrata una griglia rettangolrae e il corrispondente grafo G, uno spanning tree di G e il labirinto risultante. Per ulteriori dettagli sulla generazione di algoritmi si veda la seguente pagina
https://en.wikipedia.org/wiki/Maze_generation_algorithm.



Per realizzare il progetto si deve individuare ed implementare un qualche algoritmo di generazione di labirinti. L'algoritmo va poi utilizzato per generare labirinti di diverse dimensioni (e possibilmente anche con diverse disposizioni di celle), misurando anche i tempi di esecuzione.

12) Implementazione dell'algoritmo Dijkstra

Il progetto consiste nell'implementare l'algoritmo di Dijkstra per il calcolo dei cammini minimi da sorgente unica e nell'effettuare una valutazione sperimentale delle sue prestazioni per varie tipologie di grafi di input. L'attività sperimentale deve prevedere la generazione casuale di vari insiemi di grafi per diversi valori di n (numero di vertici) ed m (numero di archi). Più precisamente per ogni valore di densità (definita come il rapporto m/n) vanno generati un certo numero di grafi per valori crescenti di n. Per maggiori dettagli vedere il capitolo 24 del libro di testo.

13) Implementazione dell'algoritmo A*

Il progetto consiste nell'implementare l'algoritmo A* per il calcolo di un cammino minimo tra due vertici e nell'effettuare una valutazione sperimentale delle sue prestazioni per varie tipologie di grafi di input. L'algoritmo A* L'algoritmo A* opera in maniera simile all'algoritmo di Dijkstra, la differenza è che nel decidere come estendere i cammini parziali privilegia i cammini che "presumibilmente" saranno più corti. Per capire l'idea di A* si immagini di dover trovare il percorso più breve per andare tra due punti di una città. Supponiamo di trovarci ad un certo incrocio e di dover decidere quale strada prendere per continuare; senza informazioni aggiuntive dovremmo provare le varie strade che ci circondano. Se però sapessimo che la nostra destinazione si trova a nord, esploreremmo prima le strade che vanno verso nord. Ovviamente è possibile che tali strade non raggiungano la destinazione, ma in molti casi sarà così. In pratica, l'algoritmo A* sceglie il prossimo vertice da aggiungere all'albero dei cammini minimi che si sta costruendo, scegliendo il vertice v per cui è minima la quantità v.f= v.d+ v.h, dove v.d è la stima del cammino minimo da s a v (come nell'algoritmo di Dijkstra), mentre v.h è una funzione euristica che stima il costo del cammino minimo da v alla destinazione. La funzione v.h dipende dalla specifica applicazione. Un caso in cui A* presenta buone prestazioni (di solito migliori di quelle di Dijkstra) è il caso in cui si vuole cercare un cammino minimo tra due punti su una mappa (il grafo è quindi una rete stradale o simile). In questo caso si può usare come stima la distanza in linea d'aria. Per maggiori dettagli sull'algoritmo A* si veda la pagina wikipedia relativa (https://en.wikipedia.org/wiki/A*_search_algorithm).

Per realizzare il progetto si può implementare la versione dell'algoritmo A* che stima le distanze per mezzo della distanza euclidea (come nel caso delle reti stradali). In questo caso l'attività sperimentale può essere condotta generando casualmente vari grafi (possibilmente planari) i cui vertici hanno delle coordinate nel piano e i cui archi hanno un peso pari alla distanza euclidea tra i due estremi. In alternativa si possono utilizzare reti stradali reali (ad esempio, su questa pagina ci sono dati relativi alle reti stradali d'Europa

14) Implementazione di algoritmo per il calcolo dell'inviluppo convesso di un insieme di punti

L'inviluppo convesso $CH(Q)$ di un insieme di punti Q è il più piccolo poligono convesso P tale che ogni punto di Q appartiene al bordo di P o si trova al suo interno. Per semplicità assumiamo che Q contenga almeno tre punti e che i punti in Q non siano tutti collineari. Un semplice algoritmo per calcolare l'inviluppo convesso di un insieme di punti è il cosiddetto Graham's scan. Tale algoritmo costruisce l'inviluppo convesso incrementalmente considerando i punti secondo un ordine radiale e escludendo via via i punti che non appartengono all'inviluppo stesso. Per maggiori dettagli sull'algoritmo Graham's scan si veda il capitolo 33 del libro di testo.

IL progetto consiste nell'implementare l'algoritmo Graham's scan e nell'effettuare una sperimentazione sui tempi di calcolo per vari insiemi di punti di dimensione crescente.

15) Implementazione di un algoritmo sweep line per il calcolo di intersezioni tra segmenti nel piano

Dato un insieme di n segmenti nel piano si vuole determinare se ne esistono almeno due che si intersecano. L'approccio semplice a tale problema consiste nel considerare ciascuna coppia di segmenti e verificare la loro intersezione. Tale approccio ha una complessità $O(n^2)$. Scopo del presente progetto è l'implementazione di un algoritmo più efficiente basato su un approccio sweep line. Nell'approccio sweep line si immagina che una retta verticale si muova con continuità da sinistra a destra incontrando i vari segmenti. In corrispondenza di alcuni eventi notevoli (inizio e fine di un segmento) si valutano possibile intersezioni tra i segmenti intersecati in quel momento dalla retta. Per maggiori dettagli su l'algoritmo sweep line si veda il capitolo 33 del libro di testo.