Relazione sul Progetto del corso di Algoritmi

Gnocchi Marco

13 giugno 2019

Sommario

Di seguito si espone brevemente il problema la cui modellizzazione e soluzione costituiva il progetto d'esame, seguito da una descrizione degli algoritmi e strutture dati impiegati per portare a termine tali richieste, insieme ad un calcolo della complessità temporale e spaziale ad essi relativi.

Indice

1 Il Problema **3**
1.1 Dettagli aggiuntivi 3

2 La modellizzazione e le scelte **5**
2.1 I punti centrali del modello 5
2.1.1 Trovare i cammini minimi per muoversi in Città 5
2.1.2 La selezione dei Taxi 6
2.1.3 L'elaborazione degli eventi discreti 6
2.2 Sottoproblemi di magnitudine minore 8
2.2.1 Disporre i veicoli sulla rete 8
2.2.2 Il trattamento dei dati degli utenti 8
2.2.3 La stima per eccesso dei guadagni 9
2.2.4 Gli ubiqui Heap 10
2.3 Lista delle funzioni e complessità temporale e spaziale 10

3 Note **12**

1 Il Problema

Il quesito consisteva nella modellizzazione e simulazione di una rete di trasporto automatico con forti semplificazioni. Dopo aver acquisito informazioni variabili su la flotta di auto a guida autonoma a disposizione, la struttura di una rete stradale simulata, e le chiamate di una serie di clienti era necessario selezionare le vetture più adatte a evadere le singole richieste, tenendo traccia della loro autonomia, assicurandone eventualmente la ricarica, delle distanze percorse da ognuna, e del guadagno relativo alle corse. Prima della gestione della simulazione era richiesto di disporre i veicoli opportunamente sulla rete stradale, in modo da massimizzare la distanza tra essi, oltre che una breve elaborazione dei dati dei clienti.

Più nello specifico per poter andare a trattare questo problema, era necessario trovare modalità efficaci di modellizzare la rete stradale in modo che il calcolo dei cammini minimi tra un punto ed un altro fosse il più efficente possibile; stabilire regole di confronto che permettessero di selezionare, di volta in volta, il veicolo più adatto, facendo riferimento a delle specifiche fornite dal problema; e anche stabilire un metodo efficente per gestire il susseguirsi degli eventi discreti, consistenti in chiamate, ma anche corse e ricariche delle autovetture, e la loro elaborazione ordinata.

1.1 Dettagli aggiuntivi

Procedendo con maggior ordine e dettaglio, ciò che era necessario fare è quanto segue:

- elaborare oppurtunamente i parametri del problema.
- ordinare i clienti in ordine alfabetico e stamparne il nome. Ciò è fatto nell'assunzione che ogni cliente abbia un nome diverso ed effettui una sola chiamata.
- trovare durata e percorsi minimi di ognuno dei viaggi richiesti da ogni cliente e ordinarli per durata decrescente.
- stabilire la posizione iniziale delle vetture in modo che la prima sia in sede, ed ogni auto successiva venga posta nel punto della rete che massimizza la somma delle distanze minime dalle auto già fissate.
- gestire la simulazione vera e propria e stamparne i risultati:
  - Una chiamata ha un tempo minimo e massimo in cui il viaggio richiesto possono essere eseguiti: essere in grado di mandare una vettura nel punto di partenza del cliente entro il tempo minimo garantisce un bonus ai guadagni. Non è consentito terminare una corsa dopo il tempo massimo.

  - Ad ogni chiamata si sceglie l'auto in grado di terminare per prima il servizio richiesto. Se più auto terminerebbero il servizio allo stesso orario tra queste si sceglie quella che arriva dopo nel punto di partenza richiesto dal cliente, in modo da minimizzare i tempi in cui la vettura attende ferma. Se più vetture realizzano la parità anche secondo questo criterio, poichè ad ogni vettura è associato un numero univoco, si selezionerà quella di numero più basso.
  - Un auto è considerata disponibile per la corsa solo se in grado di terminarla entro il tempo massimo, se al termine avrebbe ancora carica sufficente a rientrare in sede, e se non è occupata in altre mansioni.
  - una vettura può arrivare al punto di partenza richiesto da un cliente in qualunque momento, ma non può iniziare il viaggio prima del tempo minimo. Tale tempo minimo rappresenta l'orario in cui il cliente è presente dove comunicato per il ritiro.
  - Se nessuna vettura è libera, si considera rifiutata la chiamata; è poi richiesto il numero totali di chiamate rifiutate.
  - Al termine di una corsa, se una vettura ha meno del 20% di carica, essa deve essere mandata a ricaricarsi.
  - Il tempo di ricarica di una vettura è fisso e non dipende dall'autonomia residua. Inoltre solo una vettura per volta può ricaricarsi: le altre attendono in coda. Si tiene conto del numero di ricariche.
  - La distanza totale percorsa dalle vetture durante servizio o rientro in sede per la ricarica deve essere misurata.
  - Il ricavo da una corsa è considerato uguale al tempo necessario per compiere il tragitto, più l'eventuale bonus, che è specifico della richiesta ed è comunicato al momento della chiamata. Si deve tener traccia dei ricavi totali.
  - La gestione degli eventi segue una gerarichia, in modo che, in linea di massima, i veicoli vengano liberati prima che impegnati: in questo modo, ad esempio, una vettura appena caricata è considerata libera per una chiamata giunta nel momento del termine della ricarica. L'ordine di priorità è: termine ricariche, inizio ricariche, fine corsa di una vettura e, solo in fine, l'arrivo di una chiamata.

- Compiere una stima per eccesso del guadagno che, a fronte di un orizzonte temporale di servizio limitato, scelga le chiamate di maggior valore e le serva con precedenza rispetto a quelle che porterebbero ad un minor guadagno. Qui si è fatta una grossa approssimazione fingendo che i veicoli possano prestarsi tempo a vicenda e che non abbiano bisogno di muoversi sulla rete per prendere in carico una chiamata, ma solo per evaderla.

2 La modellizzazione e le scelte

2.1 I punti centrali del modello

In questa sezione si pone l'accento su i sottoproblemi di maggiore importanza, su quali approcci sono stati scelti per modellizzarli e risolverli e su quali ragioni mi abbiano portato a selezionare tali approcci. Viene in fine indicata anche la complessità temporale e spaziale degli algoritmi implementati.

Si premette l'uso dei seguenti significanti nella notazione riguardante le complessità:

c	Numero delle chiamate
n	Numero di piazze, incroci e altri punti dove i veicoli possono fermarsi
m	Numero delle strade
a	Numero dei Taxi

2.1.1 Trovare i cammini minimi per muoversi in Città

Il problema fondante e centrale dell'elaborazione era la gestione efficace del calcolo di minime distanze su una rete stradale. Essendo questo l'unico problema direttamente dipendente dalla modellizzazione della rete, si è scelto di modellizzare quest'ultima come un grafo in cui piazze, incroci e simili sarebbero stati i nodi, e le strade gli archi, tramite l'uso di liste di adiacenza, e di impiegare l'algoritmo di Dijkstra per calcolare quelle distanze. Questo approccio è stato preferito perchè computazionalmente minimo tra le possibilità note: L'algoritmo di Dijkstra ha infatti molto da guadagnare dal più rapido accesso alla lista dei nodi adiacenti ad un nodo dato garantita dall'Implementazione per liste di adiacenza, rispetto a quella data dalle matrici di adiacenza, ed essendo implementato sfruttando gli Heap per l'identificazione del nodo più vicino a quelli già raggiunti, ha una complessità di $O(m \log n)$dove m è il numero di archi e n di vertici. Nella risoluzione del problema sono state Tuttavia impegate due diverse versioni di tale algoritmo: una che tenesse traccia dei predecessori, ricostruendo in calce il cammino compiuto, ed una versione che se ne dimenticasse. Nel primo caso alla complessità già dichiarata si aggiunge quella necessaria a ricomporre il cammino che è $O(n)$ nel caso peggiore. Osserviamo infine che in una rete stradale è piuttosto plausibile supporre che m sia più vicino a n che a n^2, suo limite superiore, poichè difficilmente esiste una strada che collega in modo diretto quasi ogni coppia di punti sulla rete. In entrambi i casi la complessità spaziale è $\Theta(n)$, con quella del secondo algoritmo maggiore della prima di circa una costante, dipendente dall'implementazione, moltiplicata per $2n$.

Si evidenzia che nell'implementazione di Dijkstra che tiene presente del percorso compiuto, si è utilizzato l'algoritmo partendo dalla destinazione invece che dal punto di partenza, in modo che il vettore dei predecessori, di cui l'algoritmo tiene naturalmente conto divenisse vettore dei successori.

Ciò è stato fatto per agevolare la ricostruzione del percorso senza conoscerne a priori la lunghezza. Tuttavia la falsa assunzione che il cammino minimo fosse unico, fatta prima di implementare in tal modo l'algoritmo, potrebbe causare differenze minime nei risultati, senza però aumentare la durata del percorso trovato. I percorsi alternativi così trovati non sono, dal punto di vista algoritmico, errati, in quanto sono comunque un elemento dell'inisieme delle soluzioni del problema, nell'istanza considerata.

2.1.2 La selezione dei Taxi

Altra questione particolarmente fondamentale per la risoluzione del problema è quella della scelta delle vetture: Per queste si è scelta una semplice implementazione in cui una struttura conteneva tutte le informazioni comuni ad ogni auto, oltre che un array con puntatori a strutture che rappresentassero le singole vetture, dotate di tutte le informazioni specifiche della singola istanza.

Per implementare questa selezione si sono fatte alcune considerazioni su come i criteri che erano proposti dalle specifiche risultassero equivalenti a criteri sul tempo di attesa della vettura nei confronti del cliente. Si è dato come tempo di idle (o di attesa della vettura) la differenza tra l'orario di arrivo del cliente nel punto di ritiro e l'orario di arrivo del taxi nel medesimo punto. Si osserva che, se vi sono vetture con tempo di idle positivo o nullo, esse realizzano necessariamente il medesimo tempo di arrivo a destinazione, che è per altro sempre inferiore al tempo di arrivo a destinazione di una qualunque vettura con tempo di idle negativo. La stessa osservazione vale anche per il premio di puntualità: una vettura è in grado di realizzarlo se e solo se ha tempo di idle non negativo. Usando delle funzioni di gestione degli heap che tenessero automaticamente in conto del terzo criterio (ovvero ordinassero le vetture con indice minore prima di quelle di indice maggiore, a parità di tempo di idle), si sono costruiti uno heap al minimo per le vetture di idle positivo, ed uno al massimo per quelle ad idle negativo. Inoltre si è verificata la disponibilità di vetture ad idle positivo prima che di vetture ad idle negativo. Questo approccio ha permesso di mantenere un'equivalenza logica tra i criteri richiesti e quelli impiegati, senza dover analizzare sempre tutte le vetture. La complessità temporale dell'algoritmo ottenuto è di $O(a^2 + am \log n)$ nel caso peggiore, mentre ha sempre una complessità spaziale dell'ordine di $\Theta(a + n)$, dato che i due heap sono sempre composti, e al limite subito distrutti.

2.1.3 L'elaborazione degli eventi discreti

Per simulare gli avvenimenti descritti dal problema si è sfruttata una simulazione ad eventi discreti, ovvero si è costruita una lista di oggetti dotati di una relazione d'ordine totale basata sull'orario e sul tipo di evento rappresen-

tato (oltre che, in casi estremi, il numero del veicolo associato, se presente) che venisse elaborata dalla testa alla coda, in cui ogni elaborazione poteva eventualmente generare a sua volta eventi. La buona posizione del modello è garantita dalla non esistenza di chiamate con orario uguale (cosa che non renderebbe più totale la relazione d'ordine fornita nelle specifiche) e dall'impossibilità di espandere all'infinito la lista di eventi, cosa garantita dal limite alla tipologia di tipi di evento generati e generabili. Per essere precisi perdere quest'ultima proprietà del problema non renderebbe mal posto il problema, quanto più non computabile in tempo finito il modello.

Per gestire una lista di eventi così implementata sono state necessarie funzioni in grado di scorrere la lista ed inserire nuovi eventi in punti qualsiasi, funzioni che permettessero di rimuovere l'elemento in testa, e in fine delle condizioni che permettessero di riconoscere il termine della coda: garantite dalla modalità di inizializzazione ed inserimento degli eventi.

Probabilmente una gestione più efficace di questa implementazione sarebbe stata, in un caso più generale, possibile utilizzando uno heap per gli eventi, ma errori valutativi durante la realizzazione del progetto, seguiti poi da considerazioni sui risultati specifici del problema, hanno spinto ad optare per una lista ordinata. Parte di queste mal poste valutazioni erano legate alla convinzione di dover realizzare uno heap pienamente dinamico nelle dimensioni, cosa che avrebbe reso difficile l'inserimento di nuovi eventi; tuttavia ogni evento diverso da una chiamata viene generato solo in sede di cancellazione di un evento, e pertanto non posso avere più di c eventi in coda: questo vincolo noto a priori avrebbe permesso di adattare facilmente la struttura di heap impiegata altrove alla coda degli eventi, rendendo l'inserimento di un nuovo evento, in posizione corretta, un'operazione possibile in tempo $O(\log c)$, mentre l'estrazione del minimo sarebbe stata possibile in tempo $\Theta(1)$, richiedendo però sempre anche $O(\log c)$ per ricomporre lo heap.

Si osserva tuttavia che la grande mole di chiamate rifiutate nel caso medio, rispetto a quelle in arrivo, rende inaspettatamente molto efficente la gestione scelta, che non richiede infatti alcuna operazione ulteriore alla valutazione, dopo all'estrazione di un evento dalla coda, ma solo $\Theta(1)$ per estrarre il massimo, e $O(c)$ per inserire un evento nella sua posizione corretta, cosa però relativamente rara in questo specifico contesto.

Nella sua complessività l'approccio implementato richiede un tempo ed uno spazio di $\Theta(c)$ per acquisire gli eventi innescati dalle chiamate, ed un tempo di $O(ca^2 + c^2 + cam \log n)$ per elaborare la totalità degli eventi, in uno spazio di $\Theta(a + n)$. In realtà proprio perchè nel caso medio, la maggior parte degli eventi è una chiamata ed essa viene rifiutata per mancanza di vetture disponibili, il tempo necessario all'esecuzione è tendenzialmente ben inferiore: se una chiamata viene ricevuta mentre nessun'auto è disponibile infatti l'algoritmo impiega un tempo $\Theta(1)$ ad evadere la chiamata ed eliminare l'evento dalla coda, contro a $O(a^2 + am \log n + c)$ necessario in caso vi siano auto disponibili.

Per via dello stretto legame tra le chiamate e gli eventi che esse innescano, a livello implementativo, si è preferito costruire una sottostruttura che rappresentasse il viaggio che il cliente richiedeva chiamando. In questo modo è stato possibile passare i dati concernenti il viaggio agli eventi generati dalle chiamate passando semplicemente il riferimento ad esso, piuttosto che copiando tutte le informazioni ad esso pertinenti.

2.2 Sottoproblemi di magnitudine minore

In seguito sono trattati diversi problemi secondari più semplici da affrontare, o ricorrenti nelle soluzioni di altre problematicità.

2.2.1 Disporre i veicoli sulla rete

Prima di avviare la simulazione è necessario disporre i veicoli lungo la rete stradale, in modo che il primo veicolo sia nella sede della compagnia di trasporti, ed ogni veicolo successivo venga posto nella piazza o incrocio che massimizza la somma delle distanze dai nodi in cui è già stata posta una vettura. Per realizzare quanto richiesto in questo caso si è deciso di costruire uno heap al massimo, in cui ogni elemento era caratterizzato da un indice, a cui era associato univocamente un nodo del grafo rappresentante la rete stradale, ed un peso su cui applicare il criterio di heap. Inizialmente questo peso corrispondeva alla distanza minima dei nodi dalla sede, la quale veniva arbitrariamente estratta, ponendovi dunque una vettura. Successivamente veniva iterato per ogni vettura rimanente il seguente processo: Si estrae il massimo dallo heap (e dunque si riconosce il nodo che realizza il criterio richiesto), si aggiunge al peso di ogni elemento rimanente dello heap la distanza minima tra il nodo estratto e quello associato a tale elemento, e, infine, si riaggiorna lo heap. Una stima per eccesso della complessità temporale è data da $O(anm \log n)$ poichè aggiorna uno heap di n elementi a volte, richiamando Dijkstra ogni volta. Lo spazio necessario è invece dell'ordine di $\Theta(n)$. Si osserva che ricostruire lo heap completamente ad ogni passaggio è necessario, poichè ad ogni passaggio i pesi vengono modificati anche di molto.

2.2.2 Il trattamento dei dati degli utenti

Le specifiche del programma richiedevano anche una piccola trattazione dei dati degli utenti, in particolare era richiesto di ordinare i clienti prima per nome, in ordine alfabetico, e poi per durata decrescente del viaggio richiesto, sempre seguendo il cammino minimo possibile. In quest'utlimo caso era chiesto di tener traccia del tragitto percorso in tali viaggi, come informazione ausiliaria.

Per il primo ordinamento si è preferito un Quicksort: algoritmo che, pur avendo un running time, nel caso peggiore di $O(c^2)$, ha un tempo di esecu-

zione, nel caso medio, dell'ordine di $c \log c$, ed è inoltre spesso più efficente di algoritmi sulla carta più efficenti, come l'heap sort, in quanto molto più efficente nell'accesso alla memoria, anche perchè non necessita di memoria di appoggio proporzionale all'input per costruire l'ordinamento, e quindi non ha bisogno di attendere che il S.O. ne riservi. In questa sezione l'ordinamento delle chiamate è stato fatto per indirizzo, per evitare di muovere più memoria del necessario.

Per il secondo si è invece sfruttato uno Heapsort, integrando ad ogni iterazione della procedura un comando di stampa: In questo modo è stato possibile stampare le chiamate nell'ordine corretto, appena selezionate dall'aggiornamento dei sottoheap coinvolti nell'ordinamento, senza dover chiamare successivamente un ulteriore ciclo. Il guadagno computazionale di questo approccio è in realtà sostanzialmente nullo, limitandosi a risparmiare sull'allocazione, inizializzazione, incremento e controllo di un contatore che permettesse di stampare le chiamate in un momento distinto da quello in cui è stato compiuto l'ordine. La procedura qui impiegata ha complessità temporale di $O(cm \log n + cn + c \log c)$, in quanto comprende anche il calcolo delle distanze e dei tragitti mediante la seconda versione di Dijkstra di cui sopra. L'ordinamento vero e proprio richiede $c \log c$. Si evidenzia che in questo caso le chiamate non sono state spostate, quanto più sono stati messi in ordine gli indici ad esse associati nello heap di supporto costruito, cosa che permette comunque di conoscere l'ordinamento richiesto.

2.2.3 La stima per eccesso dei guadagni

Infine le specifiche richiedevano di trattare costruire una stima per eccesso dei guadagni ottenibili. Come già detto la stima si avvale di grosse semplificazioni, e, nonostante ciò, è comunque ottenuta mediante un procedimento euristico, e pertanto, non garantisce la certezza di aver dato la miglire stima possibile secondo i criteri scelti.

Più in particolare si è supposto di avere un unico pool comune di tempo di servizio, e che ogni chiamata potesse essere evasa senza latenze, consumando solo il tempo necessario a compiere il tragitto richiesto, ottenendo sempre il premio di puntualità associato. L'idea è che selezionando la combinazione più redditizia tra le chiamate servibili nel tempo a disposizione, operare secondo i criteri già esposti precedentemente non avrebbe potuto portare a guadagni maggiori.

Il criterio per realizzare la stima proposta è stato la selezione, di volta in volta, delle chiamate più redditizie in un'ottica di guadagno al secondo, purchè vi fosse ancora abbastanza tempo disponibile per evaderle. In caso affermativo si agiungeva il guadagno dato dalla chiamata alla stima parziale realizzata fino a tal momento, e si sottraeva il tempo necessario a compierla dal tempo ancora disponibile.

Come già sottolineato questo procedimento è *regionevole*, ma non permette di determinare la scelta ottimale delle chiamate migliori. Tuttavia garantisce una decente approssimazione. Non è possibile, tuttavia, garantire che lo sia per eccesso. Dato tuttavia che, nel caso medio, l'ampio orizzonte temporale e il grosso numero di chiamate rifiutate, rendono il guadagno relativamente basso rispetto a quanto possibile in caso di un'organizzazione ottimale delle corse, tale stima si è sempre rivelata di molto superiore al guadagno realizzato.

2.2.4 Gli ubiqui Heap

Per moltissimi dei problemi descritti sopra si è fatto l'uso di strutture Heap, sostanzialmente ogni volta in cui è stato necessario trovare un elemento di un insieme che massimizzasse o minimizzasse qualcosa. Per questo motivo nel codice sono presenti diverse implementazioni logicamente equivalenti, con piccole differenze delle stesse funzioni di gestione degli heap. La struttura che si è implementata è composta da una sorta di struttura contenitore che tiene traccia del numero massimo ed attuale di elementi dello heap, un vettore che tiene traccia delle posizioni degli elementi, utile in caso sia necessario aggiornare un solo elemento dello heap, ed un vettore contenente le posizioni dei singoli elementi dello heap, ognuno dei quali consiste in un indice, utile per far corrispondere gli elementi dello heap a quelli di altre strutture con elementi indiciati, ed un peso, parametro su cui le varie funzioni operanti su questa struttura andranno a forzare la condizione di heap.

2.3 Lista delle funzioni e complessità temporale e spaziale

Funzione	*complessità temporale*	*complessità spaziale*
getcalls	$\Theta(c)$	$\Theta(c)$
printcalls	$\Theta(c)$	$\Theta(1)$
freecalls	$\Theta(c)$	$\Theta(1)$
printclients	$\Theta(c)$	$\Theta(1)$
Quicksort	$O(c^2)$	$\Theta(1)$
ScambiaChiamate	$\Theta(1)$	$\Theta(1)$
AddEvent	$\Theta(1)$	$\Theta(1)$
NewEvent	$\Theta(1)$	$\Theta(1)$
CallToEvent[1]	$\Theta(1)$	$\Theta(1)$
ImportaEventoChiamate	$\Theta(c)$	$\Theta(c)$
NewGraph	$\Theta(n)$	$\Theta(n)$
NuovoArcoSlegato	$\Theta(1)$	$\Theta(1)$

[1] qui ho delle chiamate a srcpy, ma ho un limite superiore piuttosto basso sulla loro lunghezza, noto a priori

Funzione	*complessità temporale*	*complessità spaziale*
NuovoArco	$\Theta(1)$	$\Theta(1)$
getgraph	$\Theta(n+m)$	$\Theta(n+m)$
newElemHeap[2]	$\Theta(1)$	$\Theta(1)$
NuovoHeap[2]	$\Theta(k)$	$\Theta(k)$
ScambiaElemHeap[2]	$\Theta(1)$	$\Theta(1)$
AggMinHeap[2] [3]	$\Theta(\log k - \log \lfloor i \rfloor)$	$\Theta(k)$
BuildMinHeap[2]	$\Theta(k)$	$\Theta(1)$
IsEmpty[2]	$\Theta(1)$	$\Theta(1)$
ExtractMin[2]	$O(\log k)$	$\Theta(1)$
AggiornaDistanza	$O(\log k)$	$\Theta(1)$
isInmHeap	$\Theta(1)$	$\Theta(1)$
FreeHeap	$\Theta(k)$	$\Theta(1)$
dijkstra	$O(m \log n)$	$\Theta(n)$
DijkTragitto	$O(m \log n + n)$	$\Theta(n)$
PrintChiamViaggio	$O(cm \log n + cn + c \log c)$	$O(c+n)$
CreaAutomobili	$\Theta(a)$	$\Theta(a)$
Rottama	$\Theta(a)$	$\Theta(1)$
PlaceCar	$O(anm \log n)$	$\Theta(n)$
StampaPosAuto	$\Theta(a)$	$\Theta(1)$
ScegliAuto	$O(a^2 + am \log n)$	$\Theta(a+n)$
ConfrontoEventi	$\Theta(1)$	$\Theta(1)$
InserisciEvento	$O(c)$	$\Theta(1)$
FineRicarica	$O(c)$	$\Theta(1)$
RientroSede	$O(c + m \log n)$	$\Theta(n)$
FineServizio	$O(c)$	$\Theta(1)$
ProssimoEvento	$\Theta(1)$	$\Theta(1)$
HandleEvent	$O(a^2 + am \log n + c)$	$\Theta(a+n)$
ElaboraListaEventi	$O(ca^2 + cam \log n + c^2)$	$\Theta(a)$
StimaGuadagnoGreedy	$O(c \log c)$	$\Theta(a+n)$

[2]Queste funzioni hanno diverse versioni di complessità uguale

[3]qui la funzione viene chiamata in una certa posizione dello heap, i. Corrisponde a idx+1 nel codice.

3 Note

Si fa presente che i calcoli della complessità ignorano le chiamate a funzioni di allocazione e deallocazione, che sono, in pessima approssimazione, considerate costanti, per uniformarle a quanto supposto per la dichiarazione statica delle variabili. Questa semplificazione è dovuta al fatto che funzioni simili hanno complessità non conoscibile a priori, che esula dall'algoritmo in cui sono inserite, poichè fanno riferimento a strutture su cui la procedura chiamante non ha controllo, e non sono note a priori oppure non sono garantite dallo standard, avendo dunque complessità dipendente dalla struttura della memoria con cui lavorano. Per motivi simili si è supposta la complessità delle funzioni di lettura (fscanf()) e scrittura (printf()) come proporzionale linearmente alla dimensione dell'input: non sono infatti pubblicate delle complessità per queste funzioni insieme alla libreria standard.

Inoltre si segnala come per il calcolo della complessità spaziale si è cercato di determinare l'ordine di grandezza del massimo spazio usato contemporaneamente, in quanto è frequente che dello spazio impiegato da un'istruzione sia liberato da un'altra. Calcolare lo spazio totale di utilizzo conteggiando come necessariamente distinto ogni bit usato in ogni dato momento dell'esecuzione sarebbe stato semplicemente troppo distante dalla realtà, in quanto avrebbe fortemente sovrastimato il quantitativo di memoria necessario all'esecuzione degli algoritmi impiegati.

Si aggiunge anche che era mia intenzione suddividere ulteriormente la libreria fornita in base alle strutture impiegate, tuttavia ciò non mi è stato possibile senza ulteriori conoscenze riguardo alla gestione di libreria con cross-referencing: argomento comunque non coperto dal corso, ed esterno ai suoi obiettivi.