Algoritmi Avanzati - Laboratorio 2

Travelling Salesman Problem

Lucchetta Bryan	Parolari Luca	Schiabel Alberto
1237584	1236601	1236598

4 giugno 2020

Indice

Elenco delle figure

Elenco delle tabelle

1 Abstract

Questo secondo homework di laboratorio di Algoritmi Avanzati ha lo scopo di implementare e confrontare algoritmi per risolvere il Traveling Salesman Problem nel caso metrico.

Gli algoritmi principali implementati sono tre:

1. Algoritmo 2-approssimato per Metric-TSP basato sul Minimum Spanning Tree. In questo caso per il calcolo dell'MST abbiamo usato l'algoritmo di Prim;
2. Algoritmo esatto di Held & Karp, che usa la programmazione dinamica;
3. Algoritmo che usa l'euristica costruttiva Closest Insertion per Metric-TSP.

Abbiamo considerato anche alcuni contributi originali rispetto agli algoritmi visti in classe; esse sono discussi e presentati nella sezione Estensioni e originalità.

Il codice è scritto in C++17 ed è opportunamente commentato per facilitarne la comprensione. Non è stata usata alcuna libreria esterna.
Le risposte alle 2 domande principali dell'homework sono riportate nella sezione Analisi dei risultati.

2 IDE e compilatore

Poiché il nostro sistema operativo di sviluppo è Windows 10, abbiamo usato l'IDE Visual Studio 2019 Community e il suo compilatore `MSVC v142 x64/x86`.

Nell'archivio allegato a questa relazione abbiamo incluso un `Makefile` per permettere la compilazione su altri sistemi operativi usando `g++-9`. Il comando da usare per la compilazione è `make all`. Nel caso la *major versione* installata di `g++` sia la 9 ma l'alias esplicito `g++-9` non esista, è possibile sovrascrivere il compilatore usato con il comando `make CXX=g++ all`.

Altri comandi sono disponibili per eseguire il benchmark degli algoritmi, oppure per eseguire semplicemente i programmi compilati. Ci si riferisca al file `README.md` incluso al progetto.

3 Benchmark

Abbiamo deciso di rendere il processo di misurazione del tempo di esecuzione dei nostri algoritmi esterno al codice dei programmi sviluppati, realizzando un script in grado di misurare i tempi di esecuzione. Per l'analisi dei dati invece abbiamo scritto uno script in Python, in grado di elaborare i risultati e produrre tabelle e grafici.

3.1 Misurazione

Il tempo di esecuzione dell'algoritmo, essendo esterno al codice, tiene conto di diversi fattori:

- Tempo necessario a leggere il file di input in un container *std::vector* temporaneo;
- Tempo per trasformare il container temporaneo in una Matrice delle Distanze;
- Tempo per eseguire l'algoritmo vero e proprio per la risoluzione di TSP e restituire il risultato;

Il processo di misurazione restituisce un file in formato CSV che riassume il risultato del benchmark. In particolare, il file contiene le seguenti colonne:

- **ms**: tempo in millisecondi per eseguire il programma su un singolo file di input;
- **output**: risultato del programma, ovvero il peso del circuito Hamiltoniano (ottimo o approssimato) individuato sul grafo dato in input;
- **d**: numero di nodi del grafo di input;
- **weight_type**: tipo delle distanze del grafo (*EUC_2D* o *GEO*), i cui valori ammissibili sono descritti in Rappresentazione del Grafo;
- **filename**: nome del file di input letto.

Per rendere i risultati del benchmark quanto più stabili e affidabili possibile, abbiamo preso le seguenti precauzioni:

- Abbiamo usato sempre lo stesso computer per misurare il tempo di esecuzione dei programmi implementati;
- Abbiamo chiuso tutti i programmi in foreground e disabilitato quanti più servizi possibile in background;
- Abbiamo disabilitato la connessione Internet del computer scelto;
- Abbiamo fatto più misurazioni in tempi differenti. Di tutte le misurazioni effettuate è poi stata scelta la minima per elaborazioni e grafici.

Il computer usato per effettuare i benchmark degli algoritmi ha le seguenti caratteristiche:

- **Sistema Operativo**: Windows 10 Education 64 bit;
- **CPU**: Intel Core i5 6600K 3.50 GHz
- **RAM**: 16 GB;

3.2 Analisi

Abbiamo definito lo script Python *benchmark/analysis.py* per automatizzare il processo di confronto degli algoritmi e la creazione di tabelle e grafici inseriti in questa relazione. Esso legge i file CSV generati dallo script di benchmark *benchmark.sh*.

Lo script trasforma i dati grezzi in dati manipolabili e li elabora estraendone le informazioni principali e mostrandole sotto forma di grafici e tabelle. Di seguito sono riportate ad alto livello le fasi eseguite dallo script:

1. Lettura di tutti i file CSV e trasformazione in DataFrames `Pandas`;
2. Esecuzione di controlli (asserzioni) sulla struttura dei dati letti e sul loro significato, per assicurare che CSV siano esenti da errori;
3. Elaborazione dei dati. In particolare i benchmark vengono raggruppati per algoritmo in una singola tabella, e per ogni riga viene mantenuto il dato con il tempo di esecuzione minore. La colonna degli output invece mantiene il valore mediano tra tutti i benchmark per ogni algoritmo.
4. Estrazione della conoscenza tramite la creazione di tabelle e grafici con semplici primitive integrate nello script.

Le primitive utilizzate nella fase 4 per la creazione di tabelle e grafici sono:

- `print_comparison`: Crea una tabella di comparativa che mostra l'errore introdotto dagli algoritmi rispetto alle soluzioni ottime.
- `plot_precision_comparison`: Crea un grafico che mostra l'errore introdotto dagli algoritmi rispetto alle soluzioni ottime.
- `plot_runtime_comparison`: Crea un grafico di che mostra il tempo di esecuzione degli algoritmi.

Le funzioni sono ampiamente documentate nello script, al quale si rimanda per ulteriori dettagli.

4 Struttura del codice

Il progetto è strutturato in un'unica soluzione Visual Studio[1] contenente molteplici progetti, uno per ogni algoritmo per il Metric-TSP implementato. Il codice di ogni progetto è contenuto nell'omonima cartella. Di seguito l'elenco dei progetti realizzati:

- **MST2Approximation**: Implementazione dell'algoritmo di 2-approssimazione basato sul Minimum Spanning Tree;
- **HeldKarp**: Implementazione dell'algoritmo esatto Held & Karp con timeout di esecuzione fissato a 2 minuti;
- **ClosestInsertion**: Implementazione dell'algoritmo di 2-approssimazione visto a lezione basato sull'euristica costruttiva Closest Insertion;
- $(*)$ **FarthestInsertion**: Implementazione dell'algoritmo di $log(n)$-approssimazione visto a lezione basato sull'euristica costruttiva Farthest Insertion;
- $(*)$ **FarthestInsertionAlternative**: Implementazione alternativa dell'algoritmo basato sull'euristica costruttiva Farthest Insertion;
- $(*)$ **SimulatedAnnealing**: Implementazione originale dell'algoritmo stocastico Simulated Annealing applicato al problema Metric-TSP, che utilizza l'euristica Nearest-Neighbor per generare la soluzione di partenza.

Abbiamo deciso di scegliere **Closest Insertion** tra le euristiche costruttive elencate nell'homework. Abbiamo anche deciso di provare l'implementazione di **Farthest Insertion** come estensione e originalità di questo homework.
I progetti indicati con $(*)$ sono delle estensioni o delle aggiunte rispetto ai 3 algoritmi inizialmente richiesti dall'homework.

La cartella *Shared* contiene le strutture dati custom e alcune classi e metodi di utilità usati condivisi tra progetti. Abbiamo configurato Visual Studio per importare automaticamente i file di header salvati nella cartella *Shared* durante la compilazione di ogni sottoprogetto. Analogamente, tale cartella è inclusa nella compilazione dal *Makefile*, grazie all'opzione *-I* del compilatore *g++*.

[1] Una soluzione Visual Studio può essere vista come un macro-progetto che contiene più sotto-moduli.

5 Scelte implementative

5.1 Rappresentazione del grafo

Gli algoritmi di questo homework operano su grafi pesati completamente connessi e non diretti. Ha quindi senso rappresentare ogni grafo con una Matrice di Adiacenza, dato che i grafi sono completi. In questa relazione useremo in maniera ambivalente i termini *Matrice di Adiacenza* e *Matrice delle Distanze*, poiché i grafi sono pesati.
I dataset contenenti i grafi di input sono in formato standard **TSPLIB** e riportano le coordinate 2D dei nodi del grafo in due possibili formati:

- **EUC_2D**: le coordinate rappresentano la posizione nello spazio euclideo a 2 dimensioni. È dunque richiesto il calcolo della distanza euclidea tra ogni nodo, con il valore arrotondato al numero intero più vicino;
- **GEO**: le coordinate rappresentano la latidudine e la longitudine di ogni punto. Il calcolo della distanza tra due punti in questo caso è più complesso, e richiede una conversione preliminari in radianti. Tale conversione avviene nel costruttore della classe `point_geo`, definita nel file `Shared/point.h`.

La formula usata per il calcolo della distanza euclidea è mostrata nel listing 1, mentre la formula usata per calcolare la distanza geodesica è illustrata nel listing 2. Si noti che tali distanze seguono i criteri di approssimazione a distanze intere definiti nella specifica dell'homework.

```cpp
// Shared/euclidean_distance.h

int euclidean_distance(const point::point_2D& i,
                       const point::point_2D& j) noexcept {
    const auto& [x_i, y_i] = i;
    const auto& [x_j, y_j] = j;

    const double x = x_i - x_j;
    const double y = y_i - y_j;

    // distanza euclidea tra i punti i e j
    const double distance = std::sqrt(std::pow(x, 2) + std::pow(y, 2));

    // arrotonda al valore intero più vicino
    return static_cast<int>(std::round(distance));
}
```

Listing 1: Funzione per il calcolo della distanza Euclidea approssimata tra due punti.

I risultati del calcolo delle distanze (*EUC_2D* o *GEO* a seconda del dataset di input) sono inseriti nella posizione corrispondente della Matrice delle Distanze. Una volta calcolate le distanze, le coordinate originali non sono mantenute: non sono infatti necessarie ai fini della rappresentazione del grafo e del calcolo della soluzione di TSP. Nella figura 1 è possbile vedere un esempio di una conversione di un grafo di esempio rappresentato da cordinate EUC_2D nella sua matrice delle distanze.

Come nel precedente homework, per semplificare la logica di indicizzazione dei nodi del grafo, la label dei nodi (originariamente numerata da 1 a n) è decrementata di 1, quindi i nodi sono rappresentati dall'intervallo numerico $[0, n-1]$.
La classe che rappresenta la Matrice delle Distanze dei grafi è definita in `DistanceMatrix.h` nella cartella *Shared*.

```cpp
// Shared/geodesic_distance.h

int geodesic_distance(const point::point_geo& i,
                      const point::point_geo& j) noexcept {
    // raggio equatoriale approssimato della terra, in km
    constexpr double RRR = 6378.388;

    const auto& [lat_i, long_i] = i;
    const auto& [lat_j, long_j] = j;

    const double q1 = std::cos(long_i - long_j);
    const double q2 = std::cos(lat_i - lat_j);
    const double q3 = std::cos(lat_i + lat_j);

    // distanza geodesica tra i punti i e j, che sono stati convertiti
    // precedentemente in radianti
    const double distance = RRR *
        std::acos(0.5 * ((1.0 + q1) * q2 - (1.0 - q1) * q3)) + 1.0;

    // ritorna la parte intera della distanza geodesica
    return static_cast<int>(std::trunc(distance));
}
```

Listing 2: Funzione per il calcolo della distanza geodesica approssimata tra due punti.

```cpp
std::vector<T> data;

// mappa la coppia di indici (row, column) della matrice in un indice per
// il vettore 1-dimensionale data. (Indicizzazione fisica)
[[nodiscard]] size_t get_index(size_t row, size_t column) const noexcept {
    return row * n_vertexes + column;
}

// valore della distanza tra i nodi (i, j). (Indicizzazione virtuale)
[[nodiscard]] T& at(size_t i, size_t j) noexcept {
    return data.at(get_index(i, j));
}
```

Listing 3: Indicizzazione virtuale e fisica della classe `DistanceMatrix.h`.

Ottimizzazioni

Poichè i grafi in esame sono non diretti ($\forall i, j \in V,\ c(i,j) = c(j,i)$), la Matrice delle Distanze è una matrice simmetrica con tutte le entry della diagonale principale pari a 0 (in quanto ogni vertice dista 0 da se stesso). Abbiamo quindi calcolato le distanze *pair-wise* solo per la parte triangolare superiore della matrice, per poi copiarle in maniera trasposta nella parte triangolare inferiore. Questo ci ha permesso quindi di evitare di calcolare le stesse distanze più volte.

La matrice è rappresentata da un singolo `std::vector`. Questo dà la garanzia che ogni riga della matrice sia definita in sezioni contigue di memoria, riduce l'overhead rispetto ad un approccio `std::vector<std::vector>`, e, nonostante la logica di indicizzazione sia un po' più complessa

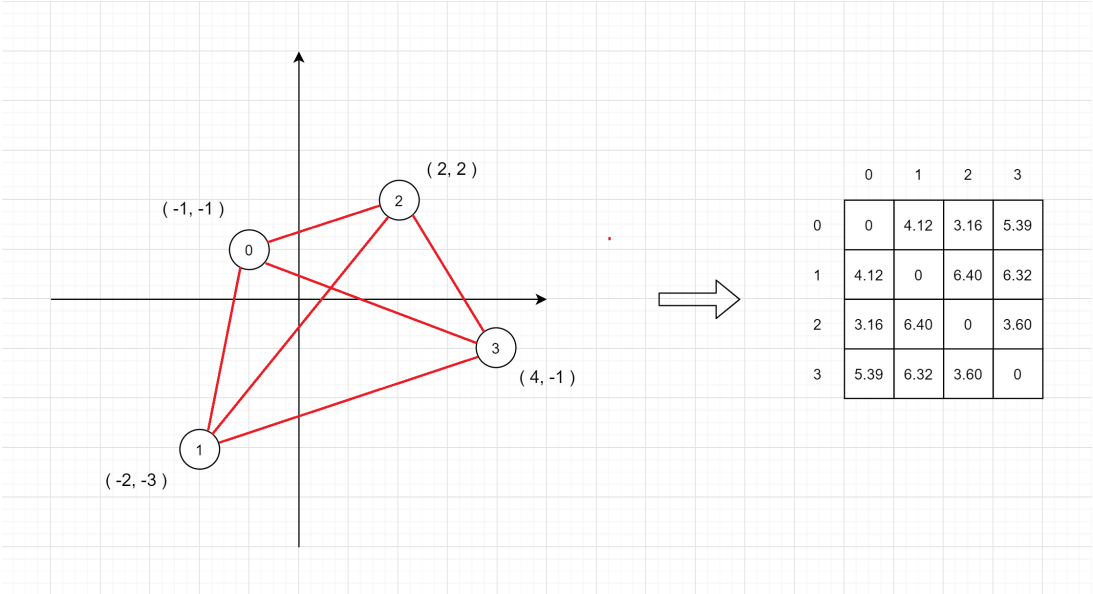

Figura 1: Grafo di esempio in uno spazio Euclideo rappresentato con Matrice delle distanze.

(l'indicizzazione virtuale è a due dimensioni, quella fisica è ad una sola dimensione), il cache behaviour della classe è migliore, dando risultati mediamente più performanti. Si veda il listing 3 per osservare la relazione tra indirizzamente fisico e virtuale nella classe `DistanceMatrix.h`.

```
int main(int argc, char** argv) {
    if (argc != 2) {
        std::cerr << "1 argument required: filename" << std::endl;
        exit(0);
    }

    const char* filename = argv[1];

    // legge il grafo completo non diretto dal file di input
    auto point_reader(read_file(filename));

    // crea la matrice di distanza, usando le distanze euclidea o geodesica a seconda
    // del tipo di input
    DistanceMatrix<int> distance_matrix = point_reader->create_distance_matrix();

    // calcola il peso della soluzione TSP individuata dall'algoritmo
    const auto total_weight = ...;

    // stampa la soluzione trovata
    std::cout << std::fixed << total_weight << std::endl;
}
```

Listing 4: Scheletro comune ad ogni file `main.cpp` del progetto.

5.2 Lettura del Grafo

Il file `main.cpp` ha la stessa struttura per ogni algoritmo, si veda il listing 4. Ad alto livello, le operazioni svolte sono:

1. Lettura del file di input: il file di input viene parsato da `read_file.h`, e vengono lette solo le informazioni più importanti, ovvero:

   - dimensione del grafo;
   - tipo di distanza (*EUC_2D* o *GEO*);
   - coordinate del grafo.

   Abbiamo usato la libreria di file streaming nativa di C++ (`fstream`). Abbiamo rappresentato il tipo di distanza con l'`enum` `EdgeWeightType.h`, per la quale abbiamo definito l'operatore di lettura `std::istream& operator>>`.

2. I punti definiti dopo la riga `NODE_COORD_SECTION` dei dataset di input sono letti con un'istanza polimorfica di `PointReader.h`, che interpreta le coordinate in maniera diversa a seconda del valore assunto dall'enumerazione `EdgeWeightType`, ovvero a seconda del tipo di distanza del file. Naturalmente, la sottoclasse `EuclideanPointReader.h` usa la distanza Euclidea, mentre `GeodesicPointReader.h` usa quella geodesica. Le classi dei punti letti in input sono definiti in `point.h` (`point_2D` per le coordinate euclidee, `point_geo` per le coordinate geografiche), e per ognuno di essi è stato definito l'operatore di lettura `std::istream& operator>>` adeguato. Questo ci ha permesso di non avere duplicazione di codice per gestire tipi diversi di punti in input.

   La label dei nodi è decrementata di 1 in questa fase di lettura.

3. Una volta letti i nodi, viene creata la matrice delle distanze applicando il *Template Method Pattern*, usando la nozione di distanza definita dalle sottoclassi di `PointReader.h`. Il metodo concreto `distance(i, j)` delle sottoclassi è usato nel costruttore di `DistanceMatrix` come *higher-order function*. Si veda il listing 5.

Tutti i file citati qui sopra sono nella cartella *Shared* del progetto consegnato e sono corredati di ulteriori commenti esplicativi.

5.3 Rappresentazione alternativa del grafo: caso MST

L'algoritmo di 2-approssimazione basato sul calcolo del Minimum Spanning Tree fa uso di due rappresentazioni diverse per i grafi. `DistanceMatrix.h` è usato per leggere il grafo in input ed eseguirne l'algoritmo di Prim (che è stato adattato dal precedente homework). `AdjacencyMapGraph.h`, che contiene un subset definite per la Mappa di Adiacenza definita nel precedente homework, è invece usata per rappresentare il Minimum Spanning Tree all'interno di `DFS.h`, che lo scorre per generare il vettore della visita *pre-order*. Questa scelta è dovuta al fatto che l'MST non è denso come il grafo di input, e anzi contiene solo $n-1$ archi. Una rappresentazione matriciale è quindi inutilmente costosa in termini di spazio in questo caso.

```cpp
// Shared/PointReader.h

class PointReader {
protected:
  std::fstream& file;
  size_t dimension;

  // calcola la distanza tra i punti i e j
  virtual int distance(size_t i, size_t j) const = 0;

public:
  PointReader(std::fstream& file, size_t dimension) :
    file(file), dimension(dimension) { }

  // distruttore virtual poiché PointReader è una classe base
  virtual ~PointReader() = default;

  // consuma la lista di coordinate dal file di input
  virtual void read() = 0;

  // crea la matrice delle distanze a partire dai punti letti. Usa il metodo distance
  // implementato dalle sotto classi come funzione higher-order
  DistanceMatrix<int> create_distance_matrix() {
    using namespace std::placeholders;
    auto distance_fun(std::bind(&PointReader::distance, this, _1, _2));

    return DistanceMatrix<int>(dimension, distance_fun);
  }
};
```

Listing 5: Definizione parziale di `Shared/PointReader.h` che evidenza la creazione della matrice delle distanze del grafo letto.

5.4 Rappresentazione dei circuiti parziali di Held & Karp

L'algoritmo Held-Karp richiede esplicitamente l'uso di due vettori:

- $d[v, S]$ è il peso del cammino minimo che parte dal nodo 0 e termina in v, visitando tutti i nodi nell'insieme S;
- $\pi[v, S]$ è il predecessore di v nel cammino minimo definito come sopra.

Poiché l'homework richiede la restituizione del peso del cammino Hamiltoniano inferiore e non il cammino stesso, abbiamo omesso il vettore $\pi[v, S]$.

Abbiamo rappresentato la tabella $d[v, S]$ usata dall'algoritmo di programmazione dinamica come una mappa chiave-valore, dove:

- la chiave è una coppia (S, v);
- il valore è il peso del cammino minimo che parte dal nodo 0 e termina in v, visitando tutti i nodi in S.

In `C++`, il tipo per rappresentare la mappa corrispondente a $d[v, S]$ è:

```
std::unordered_map<std::pair<decltype(S), size_t>, int>
```

Poiché le distanze euclidea e geodesica sono approssimate a valori interi, il tipo del valore della mappa è `int`. Per rappresentare il set di nodi S abbiamo studiato 3 possibili soluzioni.

5.4.1 Rappresentazione del circuito parziale S

Di seguito sono presentate le 3 possibili soluzioni che abbiamo individuato per rappresentare il circuito parziale S dell'algoritmo di Held & Karp. Ricordiamo che S fa parte della chiave di una `std::unordered_map`, quindi deve esistere un metodo che definisca l'hash di S.

Unordered Set

La soluzione più semplice per rappresentare un insieme di vertici senza ripetizioni è usare la struttura dati `std::unordered_set`. La tabella 1 evidenzia i pro e i contro di questo approccio. Osserviamo in particolare che lo spazio occupato per rappresentare un circuito parziale può essere compattato, il che può essere d'aiuto a limitare la RAM occupata, visto che Held & Karp ha un'occupazione spaziale esponenziale sulla taglia dell'input.

Notiamo inoltre che la libreria standard non definisce alcun metodo `std::hash<std::unordered_set<T>>`, quindi è spettata a noi la definizione della *hash function* per questa struttura dati. Si veda la sezione 5.6 per ulteriori informazioni.

PRO	CONTRO
Offerta dalla libreria standard di C++	Ha un'occupazione spaziale lineare
Rende evidente la non esistenza di nodi ripetuti	La libreria standard non ne definisce l'hash
Facile da usare	

Tabella 1: Analisi dei pro e contro di std::unordered_set.

Il tipo della mappa che rappresenta $d[v, S]$ in questo caso è:

```
std::unordered_map<std::pair<std::unordered_set<size_t>, size_t>, int>
```

Tecnica bit masking

Premessa: per questo metodo, abbiamo assunto che i programmi saranno eseguiti solo su architetture a 64 bit. I grafi con più di 63 nodi non possono essere rappresentati con questo metodo.

Una soluzione migliore per rappresentare in maniera estremamente compatta il set di nodi visitati S è usare un singolo numero intero senza segno a 64 bit. In questo caso, ogni bit a 1 rappresenta la presenza di un vertice i nell'insieme S. Se il bit all'i-esima posizione nel numero vale 1, allora il vertice $i \in S$, se il bit vale invece 0 allora $i \notin S$. La tabella 2 evidenzia i pro e i contro di questo approccio.

Nel nostro linguaggio, il tipo necessario a definire numeri interi senza segno a 64 bit è `unsigned long long`. Si consideri l'esempio in figura 2 per vedere come è possibile rappresentare S in questo modo.
Con questa rappresentazione chiamata *bit masking*, eliminazione, aggiunta e verifica della presenza di vertici in S sono implementate sfruttando operazioni `AND`, `OR`, `XOR`, bit-a-bit e bit-shifting (<<).

PRO	CONTRO
Massima compattezza spaziale	Complesso da usare
Rende evidente la non esistenza di nodi ripetuti	Supporta solo architetture a 64 bit
La libreria standard ne definisce l'hash	Usabile solo per grafi con $n \leq 63$

Tabella 2: Analisi dei pro e contro della tecnica bit masking per numeri interi senza segno.

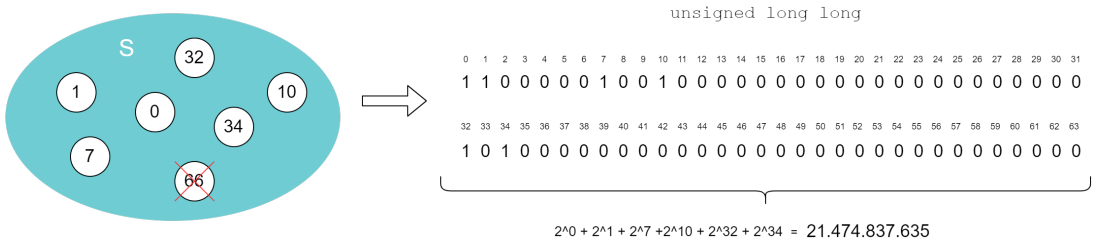

Figura 2: Rappresentazione di S tramite BitMasking

Poiché questo tipo di operazioni si traduce in una singola istruzione Assembly, manipolare l'insieme S tramite bit-masking è estremamente performante sia dal punto di vista temporale che spaziale.

Un limite di tale struttura è l'impossibilità di poter rappresentare un set S che contiene più di 64 nodi, come è possibile vedere nell'esempio della figura 2 dove il vertice 66 non può essere rappresentato in questo modo. Il limite di rappresentazione di S in realtà scende a 63 nodi per via delle operazioni di bit-shift necessarie a manipolare il circuito parziale.
Il tipo della mappa in questo caso è:

```
std::unordered_map<std::pair<unsigned long long, size_t>, int>
```

Struttura dati ad-hoc: DynamicBitMasking

Per superare il limite della rappresentazione tramite BitMasking per più di 63 nodi si è pensato di estendere l'idea del BitMasking non più ad un solo numero, ma a più numeri. In questo modo se l'insieme S può raggiungere dimensioni più grandi di 64 nodi è possibile rappresentarlo con più di un numero, dove il primo numero rappresenta i primi 64 nodi, il secondo i successivi 64 nodi e così via.

Abbiamo dunque creato un'apposita classe definita in `HeldKarp/DynamicBitMasking.h` che, tramite un `std::vector<unsigned long long>`, rappresenta un insieme S di qualunque dimensione. In questo modo, data una posizione di un bit i, è sufficiente ricavarsi l'indice del vettore dove risiede il numero che contiene il bit i (tramite la divisione intera di i con 64) e la posizione del bit all'interno del numero (tramite il resto della divisione di i con 64). Nell'esempio raffigurato in figura 3 è possibile vedere come lo stesso set S visto nell'esempio precedente ora possa essere rappresentato tramite questa nuova struttura dati.

Il tipo della mappa in questo caso è:

```
std::unordered_map<std::pair<DynamicBitMasking, size_t>, int>
```

5.4.2 Confronto tra std::unordered_set, bit masking e DynamicBitMasking

In questa sezione mostriamo un confronto tra le performance in termini di complessità spaziali e temporali delle 3 soluzioni proposte. Nel grafico riportato in figura 4 abbiamo comparato i

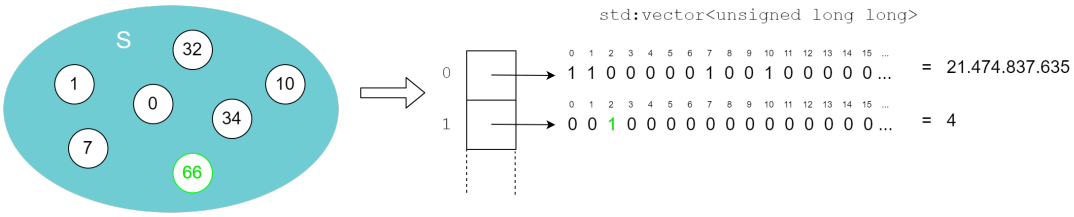

Figura 3: Rappresentazione di S tramite DynamicBitMasking

tempi d'esecuzione di HeldKarp (senza timeout) con le 3 strutture dati descritte nella sezione 5.4.1. Come input, abbiamo considerato i dataset `burma14`, `ulysses16` e `ulysses22`, aventi rispettivamente 14, 16, e 22 nodi.

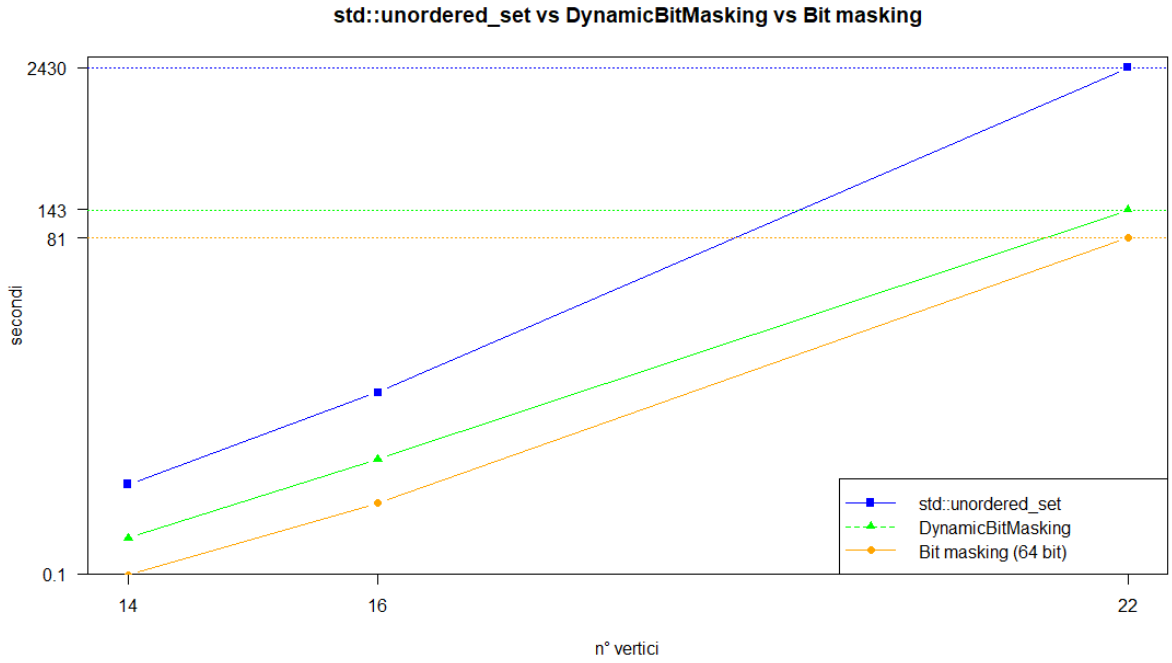

Figura 4: Confronto dei tempi di esecuzione di Held-Karp al crescere del numero di nodi usando unordered_set, BitMasking ed DynamicBitMasking. I dataset usati per il confronto sono `burma14`, `ulysses16` e `ulysses22`. Si noti che l'asse delle ordinate è in scala logaritmica.

Com'era prevedibile, l'implementazione più veloce risulta essere l'applicazione della tecnica *bit masking* che manipola un singolo numero intero senza segno a 64 bit; l'overhead delle operazioni in questo caso è infatti minimo, come minima è l'occupazione spaziale (S ha un'occupazione costante di 8 byte).

`std::unordered_set<size_t>`, al contrario, ha tempi di esecuzione più elevati, almeno un ordine di grandezza superiore rispetto alla struttura dati realizzata ad-hoc con DynamicBitMasking.

Visti i risultati sperimentali, abbiamo quindi deciso di usare BitMasking a 64 bit per implementare l'algoritmo di Held & Karp con grafi aventi meno di 64 nodi, e DynamicBitMasking per eseguire Held & Karp su grafi di dimensioni maggiori.

5.5 Timeout per Held & Karp

L'algoritmo di Held & Karp ha complessità temporale $\mathcal{O}(n^2 \cdot 2^n)$, quindi i tempi di esecuzione *esplodono* anche solo per grafi con poche decine di nodi. Come richiesto dall'homework, all'algoritmo di Held & Karp è assegnato un timeout di esecuzione T. Abbiamo fissato il valore di T a 2 minuti, per mantenere la RAM occupata sotto controllo (anche la complessità spaziale dell'algoritmo è esponenziale).

La nostra implementazione di Held & Karp, quindi:

1. Ritorna la soluzione esatta se i suoi tempi di esecuzione sono inferiori a 2 minuti, senza aspettare lo scadere del timeout;
2. Se invece il timeout scade, termina preventivamente la ricorsione e ritorna la migliore soluzione trovata fino a quel momento. A seconda della profondità del call-stack ricorsivo, la soluzione potrebbe essere ritornata qualche secondo dopo lo scadere del timeout.

C++17 non fornisce soluzione *out-of-the-box* ad alto livello per eseguire funzioni con un limite di tempo. Abbiamo quindi implementato un meccanismo di questo tipo in `Shared/timeout.h`, il cui funzionamento ad alto livello è il seguente:

- Il thread principale crea un *worker thread* incaricandolo di eseguire Held & Karp sul grafo letto in input. Fa quindi partire il timeout e resta in attesa del risultato del worker thread. Tale risultato sarà disponibile da un `std::future`.
- Se il worker thread termina prima dello scadere del timeout, il thread principale è immediatamente sbloccato e il risultato della funzione (restituito da `std::future::get()`) è ritornato al chiamante.
- Se il timeout scade e il worker thread non ha ancora terminato l'esecuzione, il thread principale gli notifica di terminare l'esecuzione il prima possibile. Tale notifica avviene forzando la conclusione di una `std::promise` creata dal main thread e data in input alla funzione eseguita incapsulata nella classe `timeout_signal`.
- Quando la funzione eseguita si accorge che il tempo a disposizione è scaduto, interrompe la ricorsione e ritorna al chiamante la migliore soluzione individuata fino a quel momento.

5.6 Hash per il set S in Held & Karp

Come spiegato in sezione 5.4.1, poiché l'insieme di nodi S visitati da Held & Karp fa parte della chiave di una `std::unordered_map`, è necessario che il tipo che rappresenta S abbia una *funzione hash* di supporto.

La funzione hash custom per `DynamicBitMasking` e per `std::unordered_set` si è ispirata alla funzione hash di `frozenset` definita nel progetto `CPython`. I dettagli della funzione sono riportati come commento in `HeldKarp/hash.h`, ma i punti focali sono:

- l'hash di partenza è un numero primo molto alto moltiplicato per un fattore dipendente da $|S|$;
- l'hash è commutativo: attraversamenti in ordini diversi di un insieme S fissato danno risultato alla stessa hash;
- l'hash dei singoli elementi di S è combinata con l'hash precedente in modo da ottenere una forte dispersione numerica, usando sia operazioni `XOR` che moltiplicazioni per numeri primi alti e diversi tra loro, la cui rappresentazione in bit è non regolare.

L'unica differenza tra la funzione hash di `DynamicBitMasking` e per `std::unordered_set` è il tipo (e di conseguenza l'iteratore) usato per attraversare $\forall x \in S$ nell'espressione:

```
for (const auto& x : S) { ... }
```

Abbiamo fatto un test delle collisioni[2] di questa hash function custom, dove abbiamo comparato il numero di collisioni ottenute con una semplice `XOR`-based hash rispetto ad una hash function definita come sopra. Come input, abbiamo usato tutte le possibili combinazioni da 1 a 16 elementi, con numeri da 0 a 15 (che rappresentano nodi di un grafo).

L'hash function banale ottiene solo 16 hash univoche e ben 32752 collisioni, mentre l'hash function più complessa citata sopra ottiene 32768 hash univoche senza alcuna collisione.

6 Algoritmi

6.1 MST2Approximation

MST2Approximation è l'implementazione dell'algoritmo di 2-approssimazione basato sul Minimum Spanning Tree visto a lezione. L'algoritmo prevede i seguenti step:

1. Selezionare un vertice radice *root* arbitrario, ad esempio 0;
2. Ricavare l'MST del grafo in input a partire da *root*, utilizzando ad esempio l'algoritmo di Prim;
3. Eseguire una visita pre-order dell'MST ricavato al passo precedente;
4. Aggiungere la radice *root* pre-order alla fine della lista ritornata dalla visita pre-order.
5. Calcolare il peso totale del circuito ricavato nei 2 passi precedenti e restituire il risultato.

Il listato 6 contiene la nostra implementazione dell'algoritmo, step per step.

```
// MST2Approximation/approx_tsp.h

// Step 1, 2
std::vector<Edge> mst(mst::prim_binary_heap_mst(distance_matrix, 0));

// Step 3
DFS dfs(std::move(mst));
const auto preorder = dfs.preorder_traversal();

// Funzione lambda che calcola la distanza tra due vertici
const auto get_distance = [&distance_matrix](const size_t x, const size_t y) {
    return distance_matrix.at(x, y);
};

// Step 4, 5
return utils::sum_weights_as_circuit(preorder.cbegin(), preorder.cend(), get_distance);
```

Listing 6: Implementazione di TSP 2-approssimato. I commenti del file originale sono stati omessi per una maggiore compattezza.

[2] https://godbolt.org/z/xdZJKX

L'algoritmo TSP 2-approssimato è stato implementato a partire dallo pseudo codice visto in classe.

6.1.1 Osservazioni

- Abbiamo usato l'algoritmo di Prim per eseguire il calcolo del Minimum Spanning Tree. Esso è infatti più adatto rispetto a Kruskal quando il grafo è rappresentato come Matrice della Distanze. Kruskal richiede di estrarre la lista di lati ordinata in modo ascendente rispetto al peso all'inizio dell'algoritmo, mentre Prim necessita solo della lista dei vertici. Ricordiamo che in un grafo completo vale l'equivalenza $m = \mathcal{O}(n^2)$.

- La coda di priorità usata dall'algoritmo di Prim è stata implementata con una Min Heap binaria, la quale è già stata descritta in dettaglio nella relazione del primo progetto.

- Come notato nella sezione Rappresentazione alternativa del grafo: caso MST, l'albero di copertura minimo è rappresentato come Mappa di Adiacenza all'interno di `Shared/DFS.h`.

- Il metodo `DFS::preorder_traversal` invoca al suo interno il metodo ricorsivo `DFS::preorder_traversal_rec`. Il listato 7 contiene la definizione di tale metodo.

- Il metodo ha tempo di esecuzione $\mathcal{O}(n+m\cdot\log(n))$, poiché l'MST è costruito in $\mathcal{O}(m\cdot\log(n))$ e la visita preorder avviene in $\mathcal{O}(n)$.

```
// Shared/DFS.h

void preorder_traversal_rec(size_t v, std::unordered_set<size_t>& visited,
                            std::vector<size_t>& path) const {
  visited.insert(v);
  path.push_back(v);

  for (const auto& [u, _] : adjacency_map.adjacent_vertexes(v)) {
    // se un nodo adiacente non è stato visitato, viene visitato
    // ricorsivamente
    if (!visited.count(u)) {
      preorder_traversal_rec(u, visited, path);
    }
  }
}
```

Listing 7: Implementazione ricorsiva della visita pre-order, inizialmente invocata sul nodo 0. I commenti del file originale sono stati omessi per una maggiore compattezza.

6.2 HeldKarp

Riportiamo lo pseudocodice dell'algortimo di programmazione dinamica Held & Karp. La funzione `HeldKarp(S, v)` vista a lezione funziona nel seguente modo:

1. Caso base 1: verifica se il percorso parziale S contenga un solo nodo, e in caso positivo restituisce la distanza tra il nodo v e il nodo di partenza 0.

2. Caso base 2: controlla se la distanza tra i nodi 0 e v, passando per tutti i nodi in S, sia già stata calcolata, e in caso positivo restituisce tale valore.

3. Caso ricorsivo:

   (a) Step a: Inizializza la distanza minima a ∞ e considera $S \setminus v$.

   (b) Step b: Scansiona tutti i nodi u in $S \setminus v = A$, calcolando ricorsivamente la distanza a partire da u e passando per tutti i vertici in A. Se la distanza trovata è inferiore a quelle precedentemente ricavate, viene sostituita.

   (c) Step c: Verifica se il timeout è scaduto. In caso positivo, effettua l'unrolling prematuro dello stack di ricorsione e ritorna il miglior risultato ottenuto fino a questo punto.

   (d) Step d: Ritorna la distanza minima calcolata del circuito parziale S.

La nostra implementazione usa due diverse strutture dati per rappresentare il sottoinsieme S tra quelle viste in 5.4.1:

- `unsigned long long` manipolati tramite BitMasking se il grafo in input ha meno di 64 nodi;
- `DynamicBitMasking` altrimenti.

Il listato 8 contiene la nostra implementazione dell'algoritmo, step per step.

6.2.1 Osservazioni

- Abbiamo voluto riportare qui la versione con BitMasking a 64 bit, la versione con DynamicBitMasking è simile. Il controllo su quale delle due implementazioni usare è fatto prima di lanciare la funzione di ricorsione appropriata.

- `signal` è il timeout descritto nella sezione Timeout per Held & Karp.

- L'algoritmo ha tempo di esecuzione $\mathcal{O}(n^2 \cdot 2^n)$ e occupazione spaziale $\mathcal{O}(n \cdot 2^n)$.

6.3 Closest Insertion

Closest Insertion è un'euristica costruttiva per Metric-TSP che consente di approssimare la soluzione ottima ad un fattore 2.

6.3.1 Idea

L'euristica è molto semplice e utilizza un'insieme di regole per scegliere il punto di partenza, il vertice da inserire ad ogni iterazione e la posizione in cui inserire il nuovo vertice. In particolare:

1. per l'inizializzazione, si considera il circuito parziale composto dal solo vertice 0; si trova un vertice j che minimizza $w(0, j)$ e si costruisce il circuito parziale $(0, j, 0)$;

2. per la selezione, si trova un vertice k non presente nel circuito parziale C che minimizza $\delta(k, C)$;

3. per l'inserimento, si trova l'arco i, j del circuito parziale che minimizza il valore $w(i, k) + w(k, j) - w(i, j)$ e lo si inserisce k tra i e j;

4. si ripete da 2 finché tutti i vertici non sono stati inseriti nel circuito.

```cpp
// HeldKarp/HeldKarp.h
using ull = unsigned long long;
using held_karp_dp_bits_t = std::unordered_map<std::pair<ull, size_t>, int>;

int held_karp_tsp_rec_bits_helper(timeout::timeout_signal& signal,
                                  DistanceMatrix<int>& distance_matrix,
                                  held_karp_dp_bits_t& C,
                                  ull bits, size_t v = 0) {
    // Caso base 1
    if (utils::is_singleton(bits, v)) {
        return distance_matrix.at(v, 0);
    }

    // Case base 2
    if (C.count({bits, v})) {
        return C[{bits, v}];
    }

    // Step a
    int min_dist = std::numeric_limits<int>::max();
    const ull difference = utils::reset_bit(bits, v);
    const size_t n = distance_matrix.size();

    // Step b
    utils::for_each(difference, n, [&](const size_t bit) {
        int dist = held_karp_tsp_rec_bits_helper(signal, distance_matrix, C,
                                                 difference, bit);
        int tmp_dist = dist + distance_matrix.at(v, bit);

        if (tmp_dist < min_dist) {
            min_dist = tmp_dist;
        }

        // Step c
        return !signal.is_expired();
    });

    // Step d
    C[{bits, v}] = min_dist;
    return min_dist;
}
```

Listing 8: Implementazione di Held e Karp con BitMasking. I commenti del file originale sono stati omessi per una maggiore compattezza.

6.3.2 Implementazione

Il listato 9 contiene la nostra implementazione dell'algoritmo.

```cpp
// ClosestInsertion/closest_insertion_tsp.h

const size_t size = distance_matrix.size();

// Funzione lambda per il calcolo della distanza tra due vertici
const auto get_distance = [&distance_matrix](const size_t x, const size_t y) {
    return distance_matrix.at(x, y);
};

// Insieme degli nodi ancora non visitati, inizialmente sono tutti i nodi.
std::unordered_set<size_t> not_visited = utils::generate_range_set(size);

// Step 1: inizializzazione
const size_t first_node = rand_int();
const size_t second_node = distance_matrix.get_closest_node(first_node);

std::vector<size_t> circuit{first_node, second_node};
circuit.reserve(size);

not_visited.erase(first_node);
not_visited.erase(second_node);

// Step 2: selezione del nodo k che minimizza d(k, circuit).
const size_t k = utils::select_new_k_minimize(not_visited, circuit, get_distance);

// Inserimento del k trovato tra due nodi i e j in modo tale da
// minimizzare la distanza totale del cammino.
circuit.emplace_back(k);
not_visited.erase(k);

// Ripetizione di selezione e inserimento finché tutti i nodi non sono stati inseriti.
while (!not_visited.empty()) {
    size_t new_k = utils::select_new_k_minimize(not_visited, circuit, get_distance);
    not_visited.erase(new_k);

    // trova l'arco che minimizza il valore di w(i, k) - w(k, j) - w(i, j)
    // e inserisce k tra i e j nel circuito
    utils::perform_best_circuit_insertion(new_k, circuit, get_distance);
}

// Restituzione la somma dei pesi del circuito trovato.
return utils::sum_weights_in_circuit(circuit.cbegin(), circuit.cend(), get_distance);
```

Listing 9: Implementazione di Closest Insertion. I commenti del file originale sono stati omessi per una maggiore compattezza.

6.3.3 Osservazioni

- All'algoritmo viene passata la matrice delle adiacenze e un random generator per la selezione del primo nodo durante l'inizializzazione. (Ulteriori dettagli in 7.2).

- L'insieme `not_visited` è utilizzato per tenere traccia degli elementi presenti nel circuito Hamiltoniano parziale: ogni volta che un vertice è inserito nel circuito, viene rimosso dall'insieme.

- L'operazione $\min w(i, j)$ è rappresentata dalla funzione

  ```
  DistanceMatrix::get_closest_node(node)
  ```

  invocabile sulla matrice di adiacenza, utilizzata nella fase di inizializzazione.

- L'operazione di minimizzazione di $\delta(k, C)$ è rappresentata dalla funzione

  ```
  utils::select_new_k_mimimize(not_visited, circuit, get_distance);
  ```

  che sceglie il vertice k che minimizza la distanza tra k ed il circuito C. Gli input a questa funzione sono l'insieme dei nodi non ancora in C da cui scegliere k, C e la funzione di distanza.

- L'inserimento del vertice selezionato è effettuato in

  ```
  utils::perform_best_circuit_insertion(new_k, circuit, get_distance);
  ```

  che sceglie la posizione di inserimento che minimizza la distanza massima del circuito.

- ClosestInsertion ha complessità temporale $\mathcal{O}(n^2)$.

- Il grafico 5 mette in relazione i dataset (individuati univocamente per numero di nodi) con l'errore percentuale di approssimazione rispetto alla soluzione esatta.

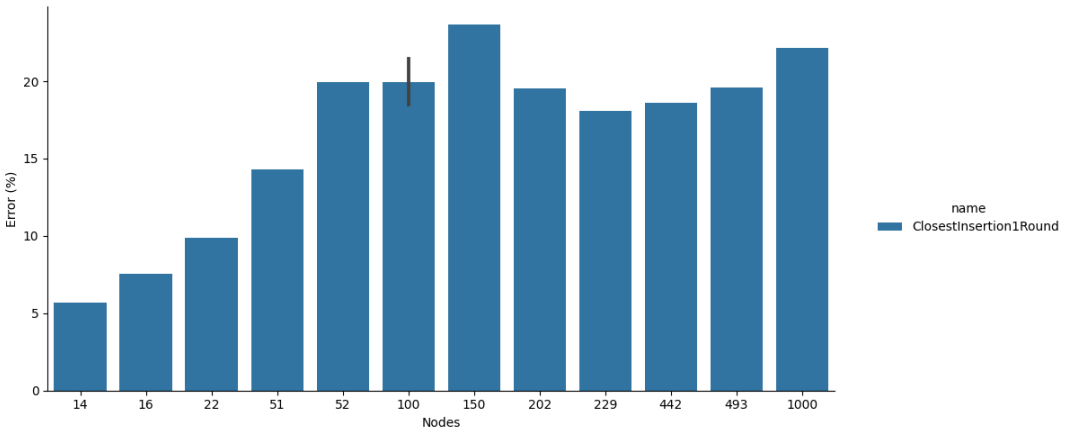

Figura 5: Errore introdotto da ClosestInsertion rispetto al numero di nodi.

7 Estensioni e originalità

Oltre alle tre implementazioni richieste dalla consegna dell'homework, abbiamo deciso di esplorare qualche altro algoritmo per il problema del commesso viaggiatore negli spazi metrici.

7.1 Farthest Insertion

Farthest Insertion è una delle euristiche impiegabili per la risoluzione di Metric-TSP in modo approssimato. Questa euristica è molto simile a Closest Insertion (si veda 6.3), infatti differisce da essa solo per la selezione del vertice k da inserire nel circuito parziale C. In particolare, in Farthest Insertion viene scelto il vertice k non presente nel circuito C che massimizza $\delta(k, C)$.

7.1.1 Implementazione

Il codice è sostanzialmente identico a quello riportato nel listato 9, con la differenza che per Farthest Insertion si effettua la scelta del nuovo nodo con:

```
utils::select_new_k_maximize(not_visited, circuit, get_distance)
```

I risultati che abbiamo ottenuto con Farthest Insertionsono illustrati dal grafico 6.

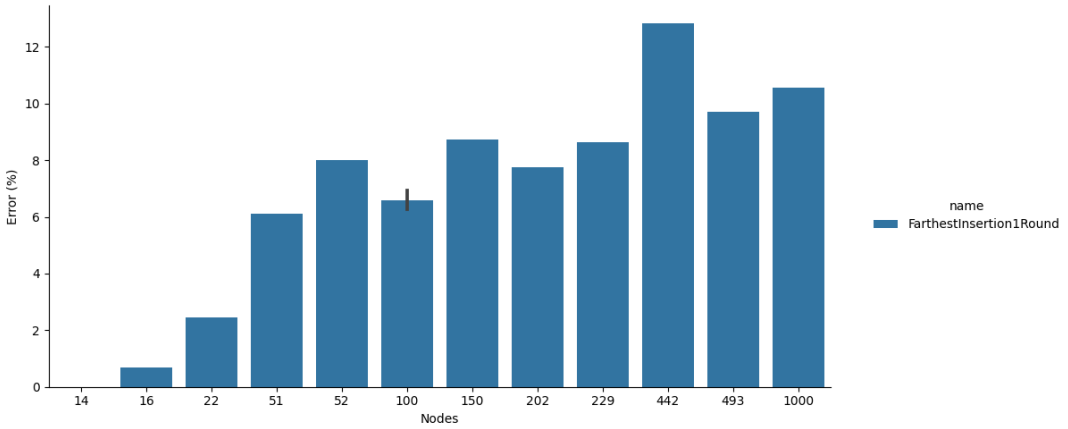

Figura 6: Errore introdotto da FarthestInsertion rispetto al numero di nodi

È doveroso mostrare le differenze tra Closest Insertion e Farthest Insertion data la dualità delle due euristiche. Come illustrato dal grafico 7, Farthest Insertion approssima meglio il problema introducendo meno errore.

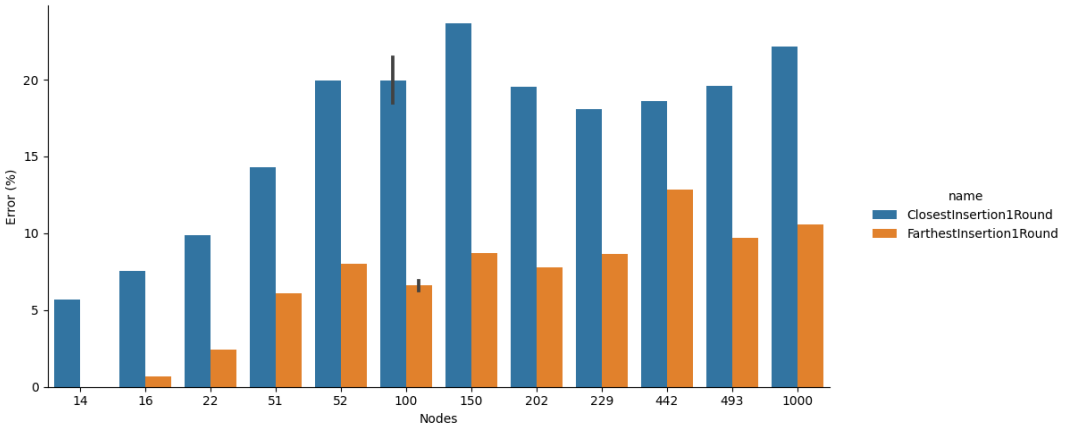

Figura 7: Confronto dell'errore introdotto da Closest Insertion e Farthest Insertion rispetto al numero di nodi

Anche per l'esecuzione con più round paralleli (meccanismo descritto in 7.2), nonostante il grafico evidenzi un leggero cambiamento dei dati, Farthest Insertion continua ad essere migliore. Si veda il grafico 8.

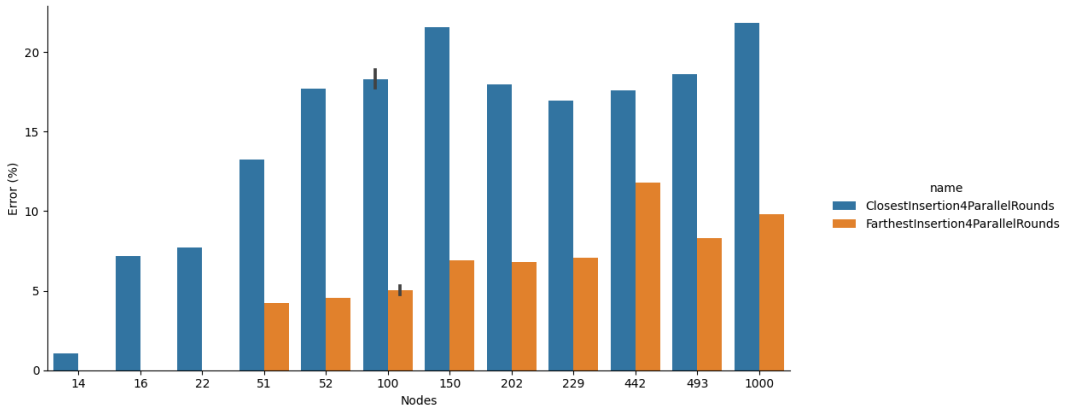

Figura 8: Confronto dell'errore introdotto da ClosestInsertion e FarthestInsertion su 4 rounds rispetto al numero di nodi

7.2 Round paralleli di algoritmi euristici sequenziali

Le euristiche costruttive come Closest Insertion (descritta alla sezione 6.3) e Farthest Insertion (descritta alla sezione 7.1) prevedono un'inizializzazione deterministica del problema, scegliendo sempre il vertice 0 come primo nodo. Tuttavia, dando la possibilità di scegliere nodi sorgenti diversi, le soluzioni ritornati dall'algoritmo potrebbero differire e molteplici esecuzioni dell'algoritmo con nodi di partenza diversi scelti arbitrariamente potrebbero produrre soluzioni più accurate rispetto ad una singola esecuzione partendo da un nodo fissato.

Abbiamo deciso di sfruttare questo fatto eseguendo più volte in parallelo l'algoritmo sequenziale con nodo iniziale casuale, e restituendo la migliore soluzione tra quelle ottenute.

Agli algoritmi è infatti passato un `RandomGenerator` che consente di generare un numero casuale, rappresentante la sorgente del ciclo Hamiltoniano da costruire. In realtà, il random generator è fornito in diverse implementazioni:

- `RealRandomGenerator`, generatore per numeri casuali di tipo reale;

- `IntegerRandomGenerator`, generatore per numeri casuali di tipo intero;
- `FixedGenerator`, generatore del numero specificato al momento della creazione del generator.

L'inizializzazione può quindi differire in base al generator fornito in input: se il generator è un `FixedGenerator` allora la sorgente scelta è fissata (come mostrato a lezione), fornendo così la versione base dell'algoritmo che non ha bisogno di molteplici esecuzioni per restituire il risultato, altrimenti la sorgente è scelta in modo casuale.

Nel nostro caso abbiamo eseguito l'algoritmo prima con sorgente fissata a 0 e poi con sorgente random e fissando il numero di istanze parallele a 4 (numero di core disponibili): i risultati dell'esecuzione parallela con nodi di partenza casuali sono stati soddisfacenti, infatti come si può notare dai grafici 9 e 10, più esecuzioni dell'algoritmo tendono a restituire soluzioni più corrette.

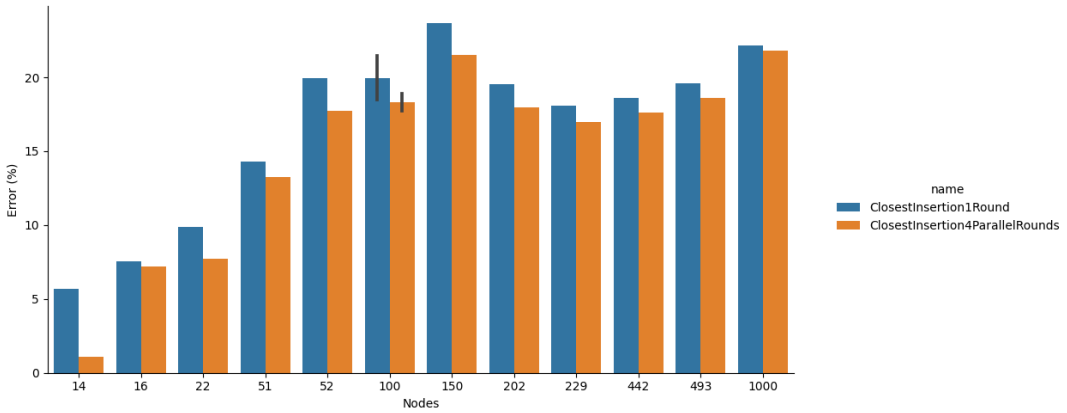

Figura 9: Confronto dell'errore introdotto da ClosestInsertion con 1 round e con 4 rounds rispetto al numero di nodi

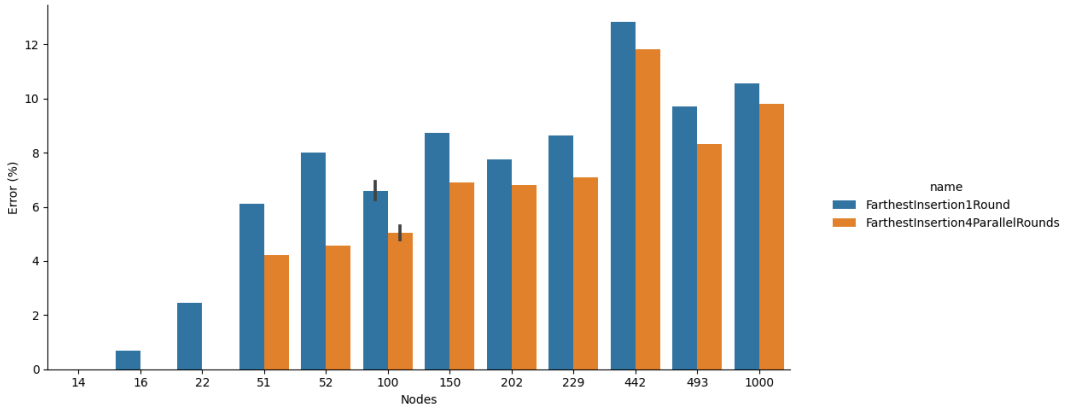

Figura 10: Confronto dell'errore introdotto da FarthestInsertion con 1 round e con 4 rounds rispetto al numero di nodi

7.3 Farthest Insertion Alternativo

Il progetto `FarthestInsertionAlternative` contiene un'implementazione alternativa dell'euristica costruttiva Farthest Insertion per risolvere il problema del TSP metrico. L'algoritmo è

tratto da *Cook et al.*[3] e prevede i seguenti step:

1. Inizializzazione: considera i due vertici u e v che compongono l'arco di costo maggiore;
2. Selezione: trova un vertice k non presente nel circuito parziale C che massimizza $\delta(k, C)$;
3. Inserimento: trova l'arco i, j del circuito parziale che minimizza il valore $w(i, k) + w(k, j) - w(i, j)$ e lo inserisce k tra i e j;
4. ripete da 2 finché non ha inserito tutti i vertici nel circuito.

Rispetto all'algoritmo descritto alla sezione 7.1, cambia solo il primo step. La complessità resta quindi $\mathcal{O}(n^2)$.

I risultati ottenuti da questo algoritmo sono illustrati dai grafici 11 e 12. Dal secondo grafico possiamo notare che rispetto alla versione standard (con un solo round) questo algoritmo si comporta un po' meglio su taglie piccole dell'input, peggiorando invece su taglie più grosse, mentre nella versione a 4 round (eseguiti su una CPU quad core in parallelo) è l'algoritmo che commette meno errori in 5 casi su 12. a competere in quasi nessun caso.

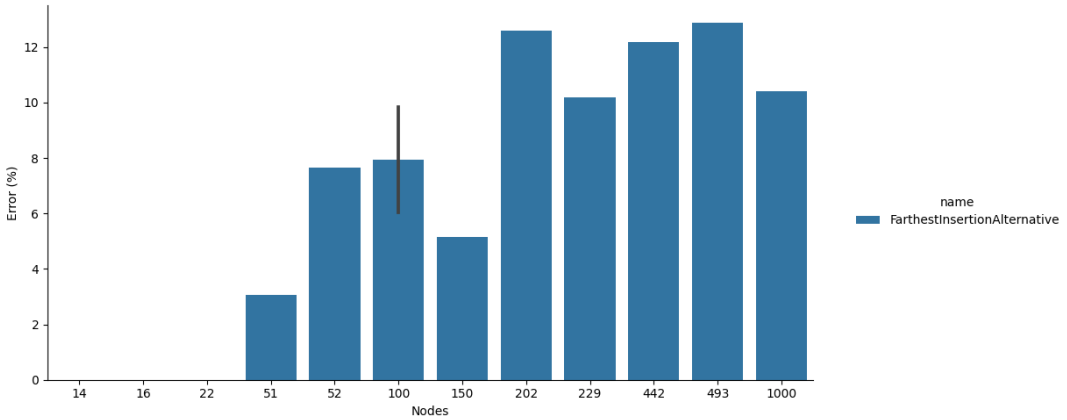

Figura 11: Errore introdotto da FarthestInsertionAlternative rispetto al numero di nodi

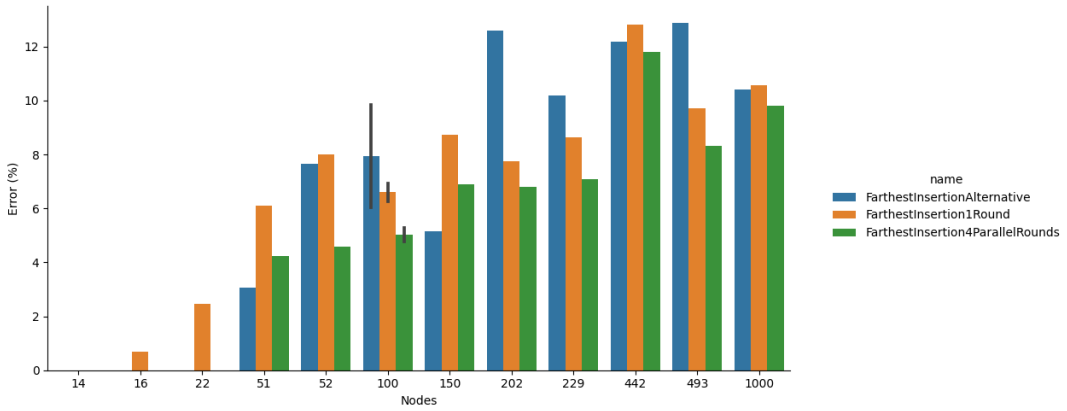

Figura 12: Confronto dell'errore introdotto da FarthestInsertionAlternative e FarthestInsertion "standard" rispetto al numero di nodi

[3] https://onlinelibrary.wiley.com/doi/10.1002/9781118033142.ch7

7.4 TSP con Simulated Annealing

7.4.1 Metodo Generale

Simulated Annealing è un metodo di ricerca stocastico in cui l'abilità di superare minimi locali è governata da un parametro di controllo detto "temperatura". Quando la temperatura è elevata, Simulated Annealing è simile ad una ricerca casuale di una soluzione, mentre a temperature basse, quando la temperatura è molto vicina a 0, l'algoritmo si comporta similmente a Gradient Descent e resta intrappolato nel minimo locale più vicino.

La temperatura è inizialmente alta, il che corrisponde ad un'alta probabilità di accettare transizioni a soluzioni non migliorative, ed è ridotta gradualmente nel tempo. La policy di raffreddamento è regolata da un altro parametro di controllo, ed emula il processo fisico di annealing, in cui un materiale solido è scaldato fino a passare allo stato liquido (dove l'energia degli atomi è massima), per essere raffreddato gradualmente per assumere una struttura cristallina (dove gli atomi tornano in uno stato di massimo ordine, e la loro energia è quindi minima).

Come altri metodi di ricerca stocastici, Simulated Annealing esplora l'universo di possibli soluzioni perturbando iterativamente una soluzione iniziale; a differenza di molti metodi, però, la temperatura dell'algoritmo permette di accettare anche soluzione peggiorative, il che aiuta ad evitare la convergenza in un minimo locale.

Ad ogni iterazione, Simulated Annealing seleziona una soluzione "vicina" alla soluzione corrente. Se la nuova soluzione ha un costo (chiamato *fitness*) migliore della precedente, è sempre accettata come nuova soluzione corrente. Se invece la nuova soluzione ha un fitness peggiore, essa è accettata con una certa probabilità (legata alla distribuzione di Boltzmann). Tale probabilità è dipendente rispetto alla differenza ΔE tra le fitness delle due soluzioni confrontate e rispetto alla temperatura corrente. La probabilità di accettare soluzioni peggiorative decresce mano a mano che la temperatura diminuisce e l'ordine di grandezza di ΔE aumenta.
Abbiamo deciso di implementare Simulated Annealing perché:

- Spesso converge a soluzioni sufficientemente vicine alla soluzione ottima in brevissimo tempo;
- È stato studiato per molti anni e la ricerca ha prodotto estensioni e miglioramenti rispetto all'algoritmo originale;
- Alcune varianti di Simulated Annealing sono già state applicate con successo a casi particolare di TSP, come ad esempio *Compressed Annealing* per risolvere *Traveling Salesman Problem with Time Windows*;
- Se si usa Random Restart, è facile da parallelizzare;
- È un algoritmo citato nel corso di Intelligenza Artificiale, ma prima d'ora non avevamo mai avuto l'occasione di implementarlo e osservarlo in pratica.

I suoi punti di debolezza, invece, sono:

- È non deterministico ed è difficile prevedere quanto la soluzione ritornata possa essere peggiore della soluzione ottima;
- Se la temperatura iniziale non è inizializzata correttamente rispetto all'input atteso, le performance dell'algoritmo degradano e le soluzioni ritornate possono essere molto distanti da quella ottima.
- Se la temperatura viene raffreddata troppo velocemente, le performance degradano similmente al punto precedente.
- Se le dimensioni degli input dell'algoritmo differiscono molto e i parametri di Simulated Annealing sono fissati, è difficile ottenere buoni risultati su tutte le istanze di input.

7.4.2 Scelta della soluzione iniziale

Simulated Annealing richiede una soluzione di partenza, la quale sarà poi iterativamente sottoposta a perturbazioni per esplorare soluzioni vicine. Nel nostro caso, abbiamo deciso di usare l'euristica **Nearest Neighbors**. Le regioni per questa scelta sono:

- È molto veloce e la soluzione ritornata non è troppo distante dalla soluzione ottima di TSP;
- È una tra le euristiche costruttive proposte nell'homework che non abbiamo implementato come metodo a sé, ed eravamo curiosi di implementarla.

Per essere ragionevolmente sicuri di partire da una buona soluzione iniziale, Nearest Neighbors è lanciato 10 volte. Di queste 16 esecuzioni, la soluzione selezionata è il circuito Hamiltoniano ritornato di peso minore.

7.4.3 Scelta delle soluzioni vicine

Il criterio di selezione di Simulated Annealing è strettamente dipendente al problema a cui è applicato. Nel caso di TSP, abbiamo deciso di generare perturbazioni in 3 modi diversi ispirati alla ricerca euristica di Lin-Kernighan[4], di cui solo uno di essi è scelto casualmente ad ogni iterazione.

Questi metodi condividono lo stesso setup iniziale, definito nella metodo `TSPSolution::manipulate_raw` in `SimulatedAnnealing/TSPSolutionPool.h`:

- Vengono selezionati casualmente due indici del circuito Hamiltoniano corrente x, y tali che $x < y$ e che $x > 0$, $y < n - 1$;
- Viene estratto un valore a caso $d \in [0, 1]$;
- Se $0 < d < 0.4$, viene eseguito uno step *2-opt*;
- Se $0.4 \leq d < 0.8$, viene eseguito uno step *translate*;
- Se $0.8 \leq d \leq 1$, viene eseguito uno step *switching*.

Chiamiamo π il circuito Hamiltoniano attuale e π' una perturbazione di tale circuito che preserva la proprietà di essere un circuito Hamiltoniano dello stesso insieme di nodi.

2-opt

1. Vengono copiati i primi x elementi di π in π';
2. I successivi x elementi di π sono copiati in ordine inverso;
3. Viene copiata l'ultima parte di π in π'.

Translate

1. Vengono copiati i primi x elementi di π in π';
2. Viene copiato l'elemento $\pi[y - 1]$ in π';
3. Vengono copiati i successivi $y - x - 1$ elementi di π in π';
4. Viene copiata l'ultima parte di π in π'.

[4] https://arxiv.org/pdf/1003.5330.pdf

Swithing

1. Vengono copiati i primi x elementi di π in π';
2. Viene copiato l'elemento $\pi[y-1]$ in π';
3. Saltando l'elemento $\pi[x]$, vengono copiati i successivi $y-x-2$ elementi di π in π';
4. Viene copiato l'elemento $\pi[x]$ in π';
5. Viene copiata l'ultima parte di π in π'.

7.4.4 Scelta della temperatura iniziale

Inizialmente avevamo fissato la temperatura inizale a 1.000.000. Questa scelta sembrava funzionare per la maggiorparte dei dataset dell'homework, ma ci siamo accorti che le performance degradavano di molto per grafi con più di 150 nodi.

Abbiamo quindi adottato il metodo di inizializzazione di Ben-Ameur[5], che si basa sul coefficiente di accettazione iniziale χ_0. χ_0 rappresenta la percentuale di transizioni sfavorevoli di simulated annealing che ci aspettiamo vengano accettate alla prima iterazione dell'algoritmo. Solitamente il valore di χ_0 è compreso nell'intervallo $[0.8, 0.\overline{99}]$. Nel nostro caso, $\chi_0 = 0.94$ ci ha dato i risultati medi migliori su tutti i dataset.

Per come abbiamo inizializzato i parametri del metodo di Ben-Ameur, grafi di dimensione più alta partiranno da una temperatura più alta, il che equivale ampliare il raggio di ricerca delle soluzioni all'aumentare della complessità dell'input.

7.4.5 Reheating

Una delle estensioni di Simulated Annealing prevede di riscaldare nuovamente la temperatura dopo un certo numero di iterazioni. L'intuizione è che questo dà la possibilità di ampliare lo spazio di ricerca delle soluzioni, e riduce maggiormente le possibilità che l'algoritmo converga in un minimo locale.

Nel nostro caso, la temperatura è aumentata ad intervalli regolari di valori via via decrescenti all'aumentare delle iterazioni di Simulated Annealing. La formula di reheating è la seguente, dove τ è la temperatura corrente, τ_0 è la temperatura iniziale, i è l'iterazione corrente, ρ è il fattore di reheating:

$$\tau = \frac{\tau_0 \cdot \rho}{10 \cdot (i+1)} \tag{1}$$

L'ampiezza degli intervalli di *reheating* è fissata e data dalla seguente formula, dove τ_0 rappresenta la temperatura iniziale:

$$max\{\frac{\tau_0}{4000}, 100\} \tag{2}$$

Tali formule sono state scelte in modo sperimentale, poiché non abbiamo trovato riferimenti a riguardo nella letteratura.

[5] https://www.researchgate.net/publication/227061666_Computing_the_Initial_Temperature_of_Simulated_Annealing

7.4.6 Parallelismo

Uno dei punti di forza di Simulated Annealing è che, se si applica Random Restart, l'algoritmo è banalmente parallelizzabile. Random Restart consiste nel lanciare un algoritmo di ricerca stocastico (nel nostro caso, Simulated Annealing) un certo numero di volte, restituendo solamente la migliore soluzione trovata.

Abbiamo definito la classe di utilità `parallel_executor` in `Shared/parallel.h`, la quale si occupa di eseguire una funzione *higher-order* su un certo numero di thread paralleli e di selezionare la migliore soluzione ottenuta. Essa è stata utilizzata per eseguire in parallelo molteplici istanze indipendenti di Simulated Annealing. In particolare, il numero di istanze eseguite in parallelo è par al numero di core fisici della CPU del computer di esecuzione.

7.4.7 Steady Steps

Per evitare il rischio di abbassare la temperatura del sistema troppo velocemente, e per esplorare un maggior numero di possibili soluzioni, l'algoritmo genera un numero costante di soluzioni vicine rispetto alla soluzione corrente prima di effettuare un passo di annealing (che consiste nel moltiplicare la temperatura corrente per il coefficiente di raffreddamento β).

Abbiamo determinato sperimentalmente che un buon numero di *steady steps* è 5.

7.4.8 Criterio di convergenza

Nella letteratura sono presenti diversi criteri per determinare quando Simulated Annealing ha effettuato un numero sufficiente di iterazione. Per cercare di garantire una certa robustezza alla nostra implementazione, ne abbiamo applicati diversi:

- **Massimo numero di iterazioni**: è il criterio più semplice, consiste nel fermarsi una volta superato un certo numero predefinito di *annealing steps*;
- **Raggiungimento temperatura minima**: l'algoritmo si ferma la temperatura ha raggiunto un valore prossimo a 0. Noi consideriamo $1 \cdot 10^{-16}$ come temperatura minima;
- **Miglior soluzione ripetuta**: la ricerca si ferma se la miglior soluzione individuata non cambia per un certo numero di iterazioni consecutive. Noi consideriamo fino a 150 ripetizioni consecutive della stessa migliore soluzione prima di dichiarare la convergenza.

Le condizioni precedentemente elencate sono trattate in modo disgiunto, ovvero l'algoritmo termina non appena si verifica almeno una delle condizioni di convergenza.

8 Analisi dei risultati

8.1 Domanda #1

> Eseguite i tre algoritmi che avete implementato (Held-Karp, euristica costruttiva e 2-approssimato) sui 13 grafi del dataset. Mostrate i risultati che avete ottenuto in una tabella come quella sottostante. Le righe della tabella corrispondono alle istanze del problema. Le colonne mostrano, per ogni algoritmo, il peso della soluzione trovata, il tempo di esecuzione e l'errore relativo calcolato come $(SoluzioneTrovata - SoluzioneOttima)/SoluzioneOttima$. Potete aggiungere altra informazione alla tabella che ritenete interessanti.

Per leggibilità la tabella richiesta è stata suddivisa nelle tabelle 3 (Held & Karp), 4 (MST 2-approssimato), 5 (Closest Insertion), una per algoritmo. Abbiamo ritenuto più pratico rappresentare i tempi in millisecondi anziché secondi, visto che la maggior parte delle rilevazioni sono tempi minori al decimo di secondo.

Held & Karp				
Istanza	Esatta	Soluzione	Tempo (ms)	Errore (%)
burma14.tsp	3323	3323	94	0
ulysses16.tsp	6859	6859	393	0
ulysses22.tsp	7013	7013	75295	0
eil51.tsp	426	986	126289	131.46
berlin52.tsp	7542	17441	126570	131.25
kroA100.tsp	21282	167464	128452	686.88
kroD100.tsp	21294	149007	128377	599.76
ch150.tsp	6528	48362	128392	640.84
gr202.tsp	40160	55127	128389	37.27
gr229.tsp	134602	176922	128344	31.44
pcb442.tsp	50778	512263	128303	908.83
d493.tsp	35002	321918	128277	819.71
dsj1000.tsp	18659688	546816520	128450	2830.47

Tabella 3: Tempo di esecuzione e errore introdotto da Held & Karp rispetto alle istanze.

Una lettura più chiara dell'errore di approssimazione è invece fornita dai grafici 13 e 14. Possiamo notare come nel secondo grafico l'errore di approssimazione di Held & Karp (con timeout) diventi praticamente impredicibile una volta superati i 22 nodi, tendendo comunque a crescere in media con la dimensione del grafo.

MST 2-approssimato				
Istanza	Esatta	Soluzione	Tempo (ms)	Errore (%)
burma14.tsp	3323	4258	34	28.14
ulysses16.tsp	6859	7857	35	14.55
ulysses22.tsp	7013	8377	34	19.45
eil51.tsp	426	563	35	32.16
berlin52.tsp	7542	10402	35	37.92
kroA100.tsp	21282	30032	36	41.11
kroD100.tsp	21294	28467	34	33.69
ch150.tsp	6528	9116	36	39.64
gr202.tsp	40160	52967	37	31.89
gr229.tsp	134602	178434	38	32.56
pcb442.tsp	50778	74254	41	46.23
d493.tsp	35002	45669	44	30.48
dsj1000.tsp	18659688	25703578	69	37.75

Tabella 4: Tempo di esecuzione e errore introdotto da MST 2-approssimato rispetto alle istanze.

Closest Insertion				
Istanza	Esatta	Soluzione	Tempo (ms)	Errore (%)
burma14.tsp	3323	3588	35	7.97
ulysses16.tsp	6859	7377	37	7.55
ulysses22.tsp	7013	7703.5	36	9.85
eil51.tsp	426	487	36	14.32
berlin52.tsp	7542	9047	37	19.95
kroA100.tsp	21282	25842	37	21.43
kroD100.tsp	21294	25230	36	18.48
ch150.tsp	6528	8072	40	23.65
gr202.tsp	40160	48011	46	19.55
gr229.tsp	134602	158933	68	18.08
pcb442.tsp	50778	60235.5	141	18.63
d493.tsp	35002	41870.5	175	19.62
dsj1000.tsp	18659688	22798432	1110	22.18

Tabella 5: Tempo di esecuzione e errore introdotto da Closest Insertion rispetto alle istanze.

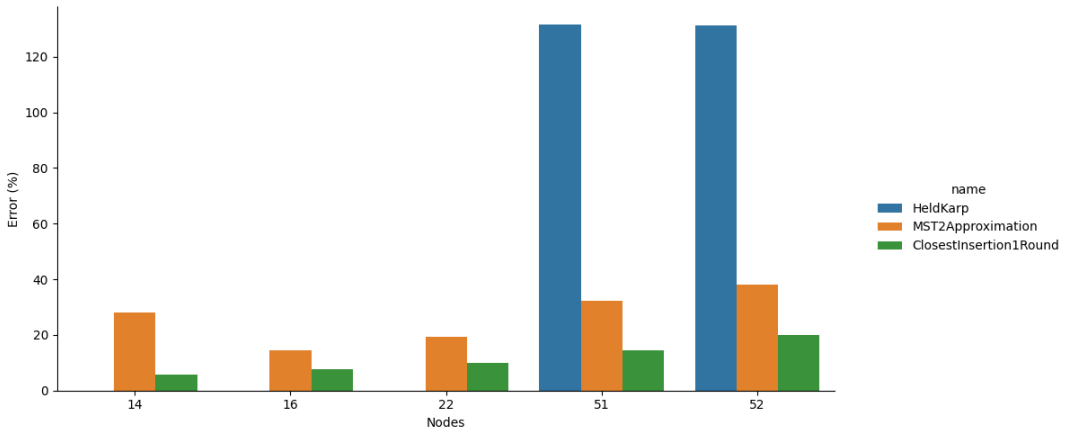

Figura 13: Confronto dell'errore introdotto da Held & Karp, MST 2-approssimato e Closest Insertion rispetto al numero di nodi (da 14 a 52)

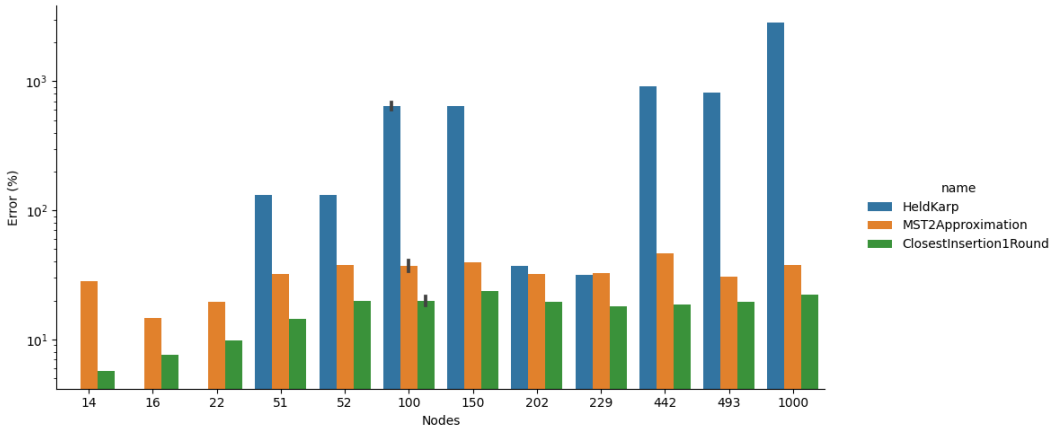

Figura 14: Confronto dell'errore introdotto da Held & Karp, MST 2-approssimato e Closest Insertion rispetto al numero di nodi (errore in scala logaritmica)

Il grafico che mostra i tempi di esecuzione per tutti e tre gli algoritmi non è stato riportato, in quanto ritenuto veramente poco informativo: Held & Karp va in timeout dopo i ventidue nodi (con il timeout fissato a due minuti), mentre il meno efficiente degli altri termina in poco più di un secondo sul grafo più grande. Abbiamo inserito quindi il grafico 15 per confrontare il runtime di MST 2-approssimato e Closest Insertion, che conferma l'efficienza in termini di tempi di esecuzione del primo.

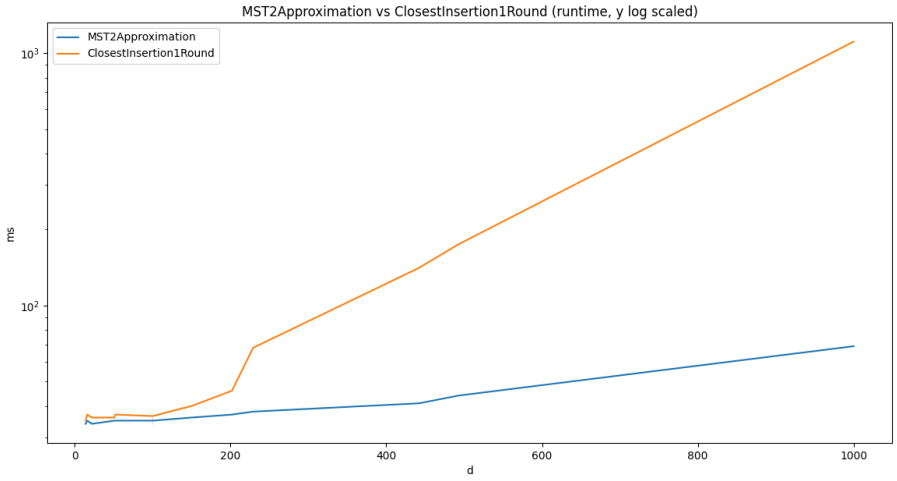

Figura 15: Confronto dei tempi di esecuzione per MST2Approximation e ClosestInsertion rispetto al numero di nodi (runtime in scala logaritmica)

8.2 Domanda #2

> Commentate i risultati che avete ottenuto: come si comportano gli algoritmi rispetti alle varie istanze? C'è un algoritmo che riesce sempre a fare meglio degli altri rispetto all'errore di approssimazione? Quale dei tre algoritmi che avete implementato è più efficiente?

Gli algoritmi che abbiamo deciso di analizzare, come scritto nella sezione 8.1, sono Held & Karp, MST 2-approssimato e Closest Insertion. I dati riportati dalle tabelle e dai grafici sono di facile lettura. Di seguito sono riportate alcune osservazioni sui risultati ottenuti.

L'algoritmo che riesce sempre a fare meglio degli altri rispetto all'errore di approssimazione è Closest Insertion. L'algoritmo riesce a fare sempre meglio di Held Karp (con timeout) e di MST 2-approssimato. L'errore introdotto è in media relativamente basso: anche per input piuttosto grandi esso rimane costante intorno al 20% circa rispetto alla soluzione ottima. Per grafi di bassa dimensione (sotto ai 100 nodi) i tempi sono comparabili a quelli di MST 2-approssimato, per poi crescere in maniera quadratica, come ci attendevamo data la complessità temporale dell'algoritmo.

L'algoritmo più efficiente dal punto di vista del tempo di esecuzione è invece MST 2-approssimato. Quest'ultimo introduce un errore di approssimazione non trascurabile anche su taglie piccole dell'input, ma compensa ciò con il tempo di esecuzione dell'algoritmo che, per dare un'idea, è inferiore ai 70 millisecondi sul grafo più grande testato. L'approssimazione introdotta, a parte l'oscillazione iniziale, rimane pressoché costante, intorno al 30/40% di errore rispetto alla soluzione ottima. Soprattutto nel caso di grafi di grossa taglia, quest'approssimazione può essere considerata soddisfacente.

Closest Insertion, anche se non è l'algoritmo più veloce dei tre, ha buoni tempi di esecuzione, e nonostante sia di 2-approssimazione come l'algoritmo che usa il Minimum Spanning Tree, sembra avere un errore medio minore in pratica.

Osservazioni

- L'algoritmo di Held & Karp, anche per taglie piccole dell'input va in timeout e sopra i 22 nodi inizia a restituire delle soluzioni parziali che si discostano molto dalla soluzione esatta. Questo fenomeno aumenta di molto soprattutto su taglie grosse dell'input, anche se è possibile notare un'inversione di tendenza per le istanze *gr202* e *gr229*, dove l'errore introdotto da Held & Karp è simile a quello di MST2Approximation. È possibile ipotizzare che questo fenomeno si verifichi per via della distribuzione dei nodi e quindi del valore che gli archi assumono, ma rimane un evento limitato, tra l'altro a due istanze la cui sorgente di dati è la stessa.

- Nella sezione 7 abbiamo descritto altri algoritmi per la risoluzione di TSP che abbiamo deciso di approfondire. In particolare, abbiamo verificato empiricamente che l'euristica costruttiva Farthest Insertion funziona meglio di Closest Insertion, e anzi, riesce a fare meglio per ogni istanza, con un'errore di approssimazione che rimane veramente basso. Per ulteriori dettagli si veda la sezione 7.1.

8.3 Altre misurazioni

Per completezza, riportiamo nelle tabelle 6, 7 e 8 i tempi e degli errori percentuali relativi agli algoritmi presentati nella sezione Estensioni e originalità.

Si osservi il curioso caso di Simulated Annealing: sembra comportarsi come un algoritmo di 2-approsimazione fino a 493 nodi, per poi esplodere nell'errore nel grafico casuale da 1000 nodi. Possiamo ipotizzare che i parametri usati in Simulated Annealing funzionino bene solo per istanze non troppo grandi. È però anche ragionevole pensare che, se `dsj1000` non fosse un grafo costruito artificialmente, forse avremmo avuto un errore di approssimazione migliore.

Farthest Insertion (standard)				
Istanza	Esatta	Soluzione	Tempo (ms)	Errore (%)
burma14.tsp	3323	3323	35	0
ulysses16.tsp	6859	6906	35	0.69
ulysses22.tsp	7013	7185	36	2.45
eil51.tsp	426	452	36	6.1
berlin52.tsp	7542	8145.5	35	8
kroA100.tsp	21282	22756	37	6.93
kroD100.tsp	21294	22627	37	6.26
ch150.tsp	6528	7097.5	41	8.72
gr202.tsp	40160	43276.5	46	7.76
gr229.tsp	134602	146236	69	8.64
pcb442.tsp	50778	57289.5	138	12.82
d493.tsp	35002	38399.5	176	9.71
dsj1000.tsp	18659688	20632741	1114	10.57

Tabella 6: Tempo di esecuzione e errore introdotto da Farthest Insertion (standard) rispetto alle istanze.

Farthest Insertion (alternativo)				
Istanza	Esatta	Soluzione	Tempo (ms)	Errore (%)
burma14.tsp	3323	3323	35	0
ulysses16.tsp	6859	6859	35	0
ulysses22.tsp	7013	7013	36	0
eil51.tsp	426	439	36	3.05
berlin52.tsp	7542	8118	36	7.64
kroA100.tsp	21282	23373	36	9.83
kroD100.tsp	21294	22577	36	6.03
ch150.tsp	6528	6864	39	5.15
gr202.tsp	40160	45211	45	12.58
gr229.tsp	134602	148324	52	10.19
pcb442.tsp	50778	56964	139	12.18
d493.tsp	35002	39506	173	12.87
dsj1000.tsp	18659688	20599827.0	1110	10.4

Tabella 7: Tempo di esecuzione e errore introdotto da Farthest Insertion (alternativo) rispetto alle istanze.

9 Conclusioni

In questa relazione abbiamo descritto gli algoritmi e relative le scelte di implementazione, tempi di esecuzione, l'errore di approssimazione e ottimizzazioni utilizzate per rendere i programmi più efficienti.

Nella sezione 8 abbiamo risposto alle 2 principali domande dell'homework, mentre nella sezione 3 abbiamo descritto il processo di benchmark adottato, pensato per essere quanto più affidabile e stabile possibile. Nelle sezioni 4, 5 e 6 abbiamo invece discusso i dettagli tecnici, quali struttura, scelte implementative e codice degli algoritmi.

La sezione 7 descrive infine le estensioni esplorate per soddisfare la nostra curiosità e arricchire il nostro bagaglio accademico, e permettendoci di esplorare alternative anche migliori.

Simulated Annealing				
Istanza	Esatta	Soluzione	Tempo (ms)	Errore (%)
burma14.tsp	3323	3854	37	15.97
ulysses16.tsp	6859	7580	37	10.51
ulysses22.tsp	7013	7802	38	11.25
eil51.tsp	426	518	38	21.6
berlin52.tsp	7542	9412	41	24.7
kroA100.tsp	21282	29467	43	38.46
kroD100.tsp	21294	28883	43	35.64
ch150.tsp	6528	9429	47	44.43
gr202.tsp	40160	54934	52	36.79
gr229.tsp	134602	198875	63	47.75
pcb442.tsp	50778	99430	86	95.81
d493.tsp	35002	61207	105	74.87
dsj1000.tsp	18659688	100466716.0	293	438.42

Tabella 8: Tempo di esecuzione e errore introdotto da Simulated Annealing rispetto alle istanze.

Questo progetto ci ha permesso di sperimentare diversi algoritmi di approssimazione ed euristiche per un problema non banale il cui algoritmo deterministico ha complessità non polinomiale. L'impossibilità di calcolare una soluzione esatta anche per dimesioni medio/grandi dell'input ci ha spinto a cercare algoritmi nuovi e migliori rispetto ai tre scelti inizialmente per l'homework.

Abbiamo infatti sperimentato nuove euristiche come Farthest Insertion e la sua implementazione alternativa; abbiamo implementato Simulated Annealing, che risolve il problema in un modo completamente diverso e che può ottenere buoni risultati, a patto che i parametri del modello siano scelti con cura.

Il progetto è disponibile anche come repository pubblica su Github:

github.com/jkomyno/algorithms-hw2