Università degli studi di
Milano-Bicocca

Decision Models

Progetto finale

The Santa's trip optimization:

approcci alternativi per determinare il percorso di consegna dei regali ottimale

Autori:
Federico Manenti - 790032 - f.manenti3@campus.unimib.it
Matteo Gaverini - 808101 - m.gaverini1@campus.unimib.it
Paolo Mariani - 800307 - p.mariani20@campus.unimib.it

Gennaio 2019

Abstract

Il problema del commesso viaggiatore o in inglese *Travelling Salesman Problem* (TSP) è un caso di studio classico che viene affrontato nell'analisi della teoria dei grafi. Questo problema oltre ad essere trattato in ambito teorico, trova applicazione in numerosi casi reali dove la logistica di un prodotto o servizio gioca un ruolo fondamentale per gli obiettivi di business. Il progetto si focalizza su una variante del TSP ambientato in uno scenario natalizio: aiutare Rudolph (la renna di Babbo Natale) ad ottimizzare il viaggio che Santa Claus deve intraprendere per consegnare i regali ai bambini di tutte le città del mondo, evitando di affaticare le renne. Per risolvere questo problema si è deciso, diversamente dalle strategie adottate comunemente in letteratura, di partire da una soluzione base generata dal *Nearest Neighbor* per poi migliorarla utilizzando delle tecniche euristiche, approcci "cluster-based" o sfruttando strumenti alternativi come *Concorde*.

1 Introduzione

Il *Travelling Salesman Problem* [1] conosciuto semplicemente come TSP, è un problema noto che si riscontra durante l'analisi di un grafo. Esso prevede di trovare, dato un insieme di nodi (città) e note le distanze tra ciascuna coppia, il cammino minimo che consente di visitare tutte le città una ed una sola volta per poi ritornare al nodo iniziale. Dal punto di vista teorico questo si traduce nell'individuazione del ciclo hamiltoniano di costo minimo presente all'interno di un grafo. Questo problema risulta essere di rilievo in numerosi casi reali dove la logistica svolge un ruolo importante in un processo decisionale. Per esempio scegliere quale è il percorso ottimale per distribuire le merci tra i magazzini oppure per consegnare i pacchi di una certa zona conoscendo a priori i destinatari a cui devono essere recapitati. Il TSP si può classificare in due tipologie: *Symmetric TSP* e *Asymmetric TSP*. Il primo prevede che la distanza tra due nodi A e B sia la stessa in entrambe le direzioni mentre nell'altro caso la distanza dipende dalla direzione con cui si percorre l'arco. Oltre alle caratteristiche definite in precedenza, si è dimostrato inoltre che il TSP appartiene alla classe dei problemi NP-hard quindi non esiste alcuna soluzione che consente di risolverlo in tempo polinomiale. Per questo motivo in letteratura sono state proposte moltissime metaeuristiche (*Simulated Annealing*, *Genetic Algorithm* etc.) che prevedono l'ottenimento di soluzioni sub-ottimali.

Il progetto si focalizza su una variante del TSP non simmetrico (*Asymmetric TSP*) ambientato in uno scenario natalizio: l'obiettivo è aiutare Rudolph (la renna di Babbo Natale) e le altre renne ad ottimizzare il viaggio che Santa Claus deve intraprendere per consegnare i regali in tutte le città del mondo (definite in una mappa) partendo dal Polo Nord per poi ritornarvi. Oltre a questo però bisogna tenere in considerazione che le renne con il passare del tempo si stancano e quindi devono rifocillarsi non solo di latte e biscotti raccolti in ogni città, ma anche di carote che si trovano esclusivamente nelle città "Prime" identificate da numeri primi. Per questo motivo se la città non risulta essere "Prime" ogni 10 passi, viene incrementato del 10% la distanza con la città successiva. Per risolvere questo problema sono state applicate diverse strategie con lo scopo di migliorare il risultato ottenuto dal *Nearest Neighbor* (NN). Inizialmente alla soluzione ottenuta dal NN, si sono applicate due classiche euristiche (*Genetic Algorithm* e *Simulated Annealing*) utilizzate in letteratura, però si è riscontrato in entrambi i casi un peggioramento della soluzione corrente. Per questo motivo si è deciso di escludere questo approccio tradizionale in favore di altri tre. Il primo prevede di utilizzare due algoritmi di *fine-tuning* (*2-Opt* e *Prime Swap*) per migliorare la soluzione ottenuta dal NN. Il secondo approccio comprende l'uso di uno strumento alternativo chiamato *Concorde*. L'ultimo approccio invece consiste nell'applicazione di una tecnica di clustering (*Gaussian Mixture*) che scomponga il problema principale in sottoproblemi individuando per ogni cluster una soluzione parziale sub-ottimale (ottenuta tramite NN, *Concorde* o risolutore *TSP Solver*) per poi unire i risultati generando la soluzione complessiva del problema. Questo approccio così come il secondo, prevede di utilizzare gli stessi algoritmi di *fine-tuning* adottati per il primo metodo.

2 Dataset

Il dataset, reso disponibile per un concorso a premi pubblicato su Kaggle, è costituito da 197.769 righe e 3 colonne (feature) così definite:

- *CityId*: identificativo della città espresso con un numero intero progressivo che parte da 0 (appartenente al Polo Nord, punto di inizio e fine del percorso).
- X: numero reale che indica la posizione della città sull'asse delle ascisse rispetto al sistema di riferimento cartesiano della mappa.

- Y: numero reale che indica la posizione della città sull'asse delle ordinate rispetto al sistema di riferimento cartesiano della mappa.

Nell'immagine 1 dell'Appendice (capitolo 7) si può osservare la distribuzione delle città.

3 Approccio Metodologico

Nel corso dell'implementazione del progetto sono stati adottati diversi approcci ed euristiche per trovare soluzioni sub-ottimali al problema in funzione del tempo e delle risorse disponibili. Inizialmente si è applicato *Nearest Neighbor* (sezione 3.1) su tutto il dataset per determinare una soluzione in maniera rapida. Nel tentativo di trovarne una migliore, si è deciso poi di applicare il solver *Concorde TSP* (sezione 3.2) che consente di ottenere soluzioni efficienti (considerando il tempo di esecuzione) caratterizzate da una distanza inferiore rispetto a quella trovata inizialmente.
Dopodiché si è cercato di migliorare ulteriormente i risultati ottenuti attraverso delle procedure di *fine-tuning* quali *Prime Swap* e *2-Opt* (sezione 3.5).
A questo punto si è cercato un metodo alternativo che potesse risolvere il problema e alla fine dopo attente riflessioni, si è giunti ad un approccio *Divide et Impera*, ovvero scomporre il problema in sottoproblemi per poi unire i risultati parziali di ciascuno di essi ottenendo la soluzione complessiva del problema originale.
Per perseguire tale strategia si è pensato inizialmente di separare la mappa in quadranti, il metodo però si è rivelato poco efficiente; si è perciò adottato un secondo approccio che prevede il clustering delle città (sezione 3.4). Le soluzioni parziali ottenute in ogni cluster si ricavano con *Nearest Neighbor*, *Concorde* oppure con il risolutore *TSP Solver* (sezione 3.3).

3.1 Nearest Neighbor

Il *Nearest Neighbor* [2] è un algoritmo che permette di trovare una soluzione sub-ottimale al TSP. A partire da una città iniziale i appartenente ad un insieme I (lista delle città da visitare), l'obiettivo è trovare la città più vicina a i, aggiungerla al percorso che si sta realizzando e rimuoverla dall'insieme I. Dopodichè tale procedimento iterativo si ripete con l'ultima città inserita

nel percorso fino a quando tutti gli elementi di I sono stati rimossi (si veda pseudocodice *Algorithm* 1).

Algorithm 1 Nearest Neighbor

1: **procedure** NEAREST NEIGHBOR(elenco città I con relative distanze)
2: $sequenza \leftarrow PoloNord$
3: **while** lunghezza(I) > 0 **do**
4: Selezionare città j appartenente a I che è più vicina all'ultima città inserita nella *sequenza*
5: Aggiungere j alla *sequenza*
6: Eliminare j dall'insieme I
7: Aggiungere la città $PoloNord$ alla *sequenza* per creare il ciclo di visita delle città
8: **return** *sequenza*

Per quanto riguarda l'algoritmo implementato, si osserva che sia la città iniziale e finale della sequenza coincide con il Polo Nord (vincolo imposto dal problema). Questa situazione però non avviene nei casi in cui si applica una tecnica di clustering, infatti una volta ottenuti i cluster si deve ottenere per ciascuno di essi una sequenza minima dove la città iniziale e finale risulta essere differente dal Polo Nord. Per questo motivo il *Nearest Neighbor*, applicato su ogni cluster, prevede delle modifiche implementative sulla scelta dei punti iniziali della sequenza. Per decretare la città più vicina si è utilizzata la distanza *euclidea* non tenendo conto delle possibili penalità imposte dal problema; tali sanzioni sono state considerate solamente in un secondo momento per calcolare la distanza complessiva del percorso.

3.2 Concorde

Il *Concorde* è un risolutore implementato in C che permette di ottenere in tempi molto rapidi una soluzione sub-ottimale al problema TSP. La sua efficienza e velocità è legata al fatto che utilizza in maniera combinata due algoritmi che consentono di perfezionare il processo di scoperta della sequenza sub-ottimale. Essi sono:

- **Chained Lin-Kernighan** [3]: algoritmo di ottimizzazione locale che, applicato alle sequenze, permette di creare un albero di soluzioni dal

processo di ottimizzazione, differenziandole rispetto a quanti e quali *swap* (scambi) si applicano all'interno della sequenza.

- **Branch & Cut** [4]: algoritmo nato negli anni 90 come unione di *Branch and Bound* e *Cutting Planes*, è utilizzato per scegliere dall'albero di soluzioni le sequenze migliori e più promettenti su cui verrà applicato nuovamente l'algoritmo *Lin-Kernighan* con un *kick* (inizio di una nuova sessione di ottimizzazione a partire dalla soluzione precedente). Determina quindi i rami dell'albero delle soluzioni da tagliare poiché non promettenti. Grazie a questo meccanismo il tempo di esecuzione dell'algoritmo è estremamente ridotto rispetto ad altre euristiche alternative per l'ottimizzazione di sequenze (es. *k-Opt*).

Nella fase iniziale della sua esecuzione il solver determina una sequenza randomica, a partire da questa applica l'algoritmo di *Chained Lin-Kernighan* in modo da individuare la combinazione di *swap* necessari per decretare una soluzione migliore. Questa operazione però non viene effettuata su tutta la sequenza, ma solamente su porzioni (*sub-tour*) concedendo ad ogni step di ottimizzazione, la possibilità di peggiorare il percorso locale intermedio per migliorare la soluzione globale (diversamente da *2-Opt* e *3-Opt*).
L'algoritmo agisce in maniera adattiva: determina ad ogni passo quanti *swap* devono essere effettuati nella sequenza tramite funzioni di costo associate ad ogni scambio e al meccanismo di selezione dei rami promettenti implementato da *Branch & Cut*. Se al termine di ogni sua esecuzione trova un risultato migliore di quello precedente, sostituisce la soluzione ottima ottenuta finora con quella appena trovata, altrimenti la scarta. Questo processo viene ripetuto con *kick* successivi fino a quando la sequenza non può essere ulteriormente migliorata. Il punto di forza di questo algoritmo rispetto ad euristiche e solver alternativi è la sua velocità di esecuzione: utilizzando un albero di soluzioni si riescono subito a determinare i cammini più promettenti quindi si riduce il tempo necessario per determinare la soluzione.
Anche in questo caso così come nel *Nearest Neighbor*, non viene inclusa nell'algoritmo la penalizzazione prevista dal problema per calcolare il percorso, ma viene misurata successivamente.

3.3 TSP Solver

Il *TSP Solver* sfrutta un algoritmo di tipo *greedy* per determinare il cammino sub-ottimale che deve essere percorso [5]. A partire da un insieme di nodi

I presenti all'interno di un grafo dove ognuno appartiene ad un frammento (*snippet*) unitario, l'obiettivo è trovare i due frammenti i e j più vicini (in base alla distanza *euclidea* dei suoi punti estremi) per poi collegarli formando un nuovo percorso contenuto in un unico *snippet*. Questo procedimento viene ripetuto fino a quando si ha un solo frammento unitario (si veda pseudocodice *Algorithm* 2).

Algorithm 2 TSP Solver

```
1: procedure TSP Solver(elenco città I con relative distanze)
2:     for each nodo in I do: Creazione frammento unitario per ogni nodo
3:     while numero frammenti > 2 do:
4:         Ricerca dei due frammenti i e j più vicini
5:         Unione di i e j in un unico frammento.
6:     return frammento_complessivo
```

L'algoritmo possiede una complessità polinomiale e spesso produce soluzioni lontane dall'ottimo globale. Per questo motivo il solver utilizza un processo di ottimizzazione che prevede di effettuare lo scambio dei punti seguendo l'ordine della sequenza. L'algoritmo viene utilizzato perché diversamente dal *Nearest Neighbor*, permette di specificare la città di partenza e arrivo del percorso, operazione necessaria quando si adotta l'approccio *Divide et Impera* nel caso in esame.

3.4 Clustering

Il clustering è una metodologia che prevede di raggruppare le osservazioni in determinati gruppi sulla base di una misura di similarità o distanza [6]. Nel caso in esame questa tecnica viene utilizzata nell'approccio *Divide et Impera*. Per far ciò si è reso necessario scegliere la tipologia di algoritmo di clustering che, dopo diverse valutazioni, si è rivelato essere il *Gaussian Mixture*. Si è scelto questo algoritmo perchè rispetto agli altri candidati tra i quali *K-Means* e *DBSCAN*, ha permesso di ottenere risultati più soddisfacenti. Successivamente si è decretato il numero k di cluster da ottenere. Per far ciò si è applicato *AutoML* [7], un processo che permette di ottimizzare una funzione obiettivo determinando la combinazione ideale degli iperparametri (*Hyperparameter Optimization*). Considerando il contesto del problema, si

vuole minimizzare la funzione che restituisce il valore della distanza calcolata con numero k di cluster.
Per effettuare *AutoML* è stata sfruttata *pyGPGO*, una libreria *python* che consente di effettuare un'ottimizzazione bayesiana tramite diversi modelli surrogati (tra i quali *Gaussian Process* e *Random Forest*).
Nel caso in esame si è utilizzato come modello surrogato *Random Forest*, dato che l'iperparametro k rappresenta una grandezza intera, mentre come funzione di acquisizione si è scelta *Expected Improvement*. Per minimizzare la funzione obiettivo si sono proposti due metodi che seguendo strade diverse, cercano di perseguire lo stesso scopo.

3.4.1 Prima proposta

Il primo metodo prevede, dopo aver effettuato clustering delle città, il calcolo dei centroidi, successivamente si determina l'ordine di visita dei cluster applicando *Nearest Neighbor* o *Concorde* ai centroidi. Ciò permette di capire in che ordine i cluster dovranno essere visitati, ma non l'ordine di percorrenza delle città all'interno di essi.
La procedura di creazione della sequenza ha inizio individuando il cluster contenente il Polo Nord (nodo da cui partire) per poi determinare l'ordine di visita delle sue città. In seguito la sequenza restituita viene ampliata annettendo, in funzione dell'ordine di visita dei cluster (calcolato tramite i centroidi), le sequenze sub-ottimali ottenute per ciascun raggruppamento collegate tramite il percorso minimo che è identificato tra ogni coppia di cluster contigui (determinato con il calcolo delle due città più vicine dei due cluster presi in considerazione).
Una volta ricavata la sequenza finale, si applicano poi le procedure di *fine-tuning* presentate nella sezione 3.5, per migliorare la soluzione ottenuta.
Si può tradurre il procedimento con lo pseudocodice 3 situato nell'Appendice (Capitolo 7).

3.4.2 Seconda Proposta

Nel secondo metodo invece differisce il modo in cui si stabilisce l'ordine di visita dei cluster e di conseguenza il percorso ottimo. Esso non prevede più di calcolare i centroidi utilizzati nel primo metodo, ma individua nodi estremi per ogni gruppo sfruttando il concetto di *punto cardinale*. L'idea è che per ogni cluster si identificano i 2 punti più estremi per ogni coordinata

(Nord, Sud, Ovest, Est). A questo punto gli 8 estremi che rappresentano i vari cluster vengono utilizzati iterativamente per realizzare una matrice di distanza, dove le righe rappresentano gli estremi del cluster che si sta considerando (inizialmente è quello a cui appartiene il Polo Nord) mentre le colonne quelli di tutti gli altri gruppi (cluster non ancora visitati). Alla fine da questa matrice si ottiene la coppia (a, b) di nodi che rappresenta la distanza minima tra un estremo del cluster che si sta considerando (gruppo a cui appartiene a) e un qualsiasi altro punto estremo di un cluster diverso (gruppo a cui appartiene b). Quindi la coppia (a, b) non solo indica il "link minimo" per passare da un cluster ad un altro, ma ne decreta anche l'ordine di visita. A questo punto, una volta individuato il collegamento, si rimuove il cluster dalla lista di quelli da visitare e si passa a quello successivo dove si reitera il procedimento. Questo passaggio, fondamentale nell'approccio che si sta descrivendo, si ripete fino a quando tutti i cluster sono stati visitati.
Una volta individuato l'ordine di visita dei gruppi e i link, si applica per ogni cluster l'algoritmo *TSP Solver* impostando come nodo iniziale e finale gli estremi di due link che corrispondono allo stesso cluster. Questa operazione avviene per tutti i link meno per il primo e l'ultimo dove il nodo iniziale per il primo collegamento e il nodo finale per l'altro viene definito a priori assegnando il Polo Nord. Successivamente tutte le sequenze sub-ottimali dei cluster vengono unite sfruttando i link individuati nella prima fase; si ottiene così un'unica sequenza finale che poi viene raffinata attraverso procedure di *fine-tuning* presentate nella sezione 3.5. Si può rappresentare il processo appena descritto con lo pseudocodice 6 nella sezione Appendice 7.

3.5 Fine-tuning

3.5.1 Prime Swap

Il *Prime Swap* è un algoritmo di *fine-tuning* il cui obiettivo è quello di minimizzare l'effetto peggiorativo causato dalle penalità a cui è sottoposto Santa Claus. Esso prevede lo scorrimento della sequenza per determinare degli scambi tra città "Prime" e non "Prime" in posizioni strategiche: ogni volta che incontra nel cammino una città con *CityId* primo non posizionata dopo il decimo passo, valuta se conviene scambiarla con le due città precedenti o le tre successive che causano l'aggiunta della penalità in funzione della loro posizione (il loro indice posizionale nella sequenza termina con la cifra 9). Lo *swap* avviene solo nel momento in cui la porzione della sequenza modificata

porti un miglioramento. La valutazione del cambiamento non avviene considerando l'intera sequenza ma solamente la porzione ridotta, garantendo un processo efficiente di ottimizzazione.

3.5.2 2-Opt

Il *2-Opt* è un algoritmo euristico di ricerca locale proposto da *G.A. Croes* [8] nel 1958 per risolvere il problema del TSP. Tale algoritmo migliora incrementalmente una sequenza iniziale attraverso dei raffinamenti successivi che prevedono lo scambio tra due archi presenti nel cammino. Nello specifico l'algoritmo in ogni step migliorativo, seleziona due archi $(u1, u2)$ e $(v1, v2)$ tale che i vertici $u1, u2, v1, v2$ appaiano in questo ordine, successivamente scambia i nodi dei due archi ottenendone due nuovi $(u1, v1)$ e $(u2, v2)$ e alla fine verifica se la nuova soluzione risulta essere migliore di quella corrente: se è cosi il nuovo percorso include gli archi ottenuti, altrimenti rimangono gli archi originali.

Inizialmente si è pensato di utilizzare entrambe le tecniche di *fine-tuning* presentate, però alla fine si è preferito usare solamente il *Prime Swap*. Il *2-Opt* è stato scartato perché valuta tutti i possibili scambi all'interno della sequenza quindi computazionalmente più oneroso.

4 Risultati e Valutazione

Prima di presentare i risultati, si vuole sottolineare che per confrontare diverse euristiche, a parità di condizioni con esiti costanti nel tempo, si sono scelti dei *seed* garantendo la replicabilità delle soluzioni a fronte di esecuzioni diverse. La ripetizione dell'esperimento con *seed* diversi permette di ottenere una componente di variabilità nel processo di selezione del miglior percorso. Tutti gli esperimenti sono stati eseguiti sfruttando le macchine virtuali messe a disposizione da Google con il servizio *Google Colab*. Ogni macchina dispone 25 GB di RAM e 68 GB di disco fisso. Di seguito vengono presentati i risultati raggiunti (si veda Tabella 1), inoltre viene indicata la soluzione migliore presentata sulla piattaforma Kaggle che risulta essere 1 513 747,36.

Table 1: Risultati

Algoritmo	**Parametri** (*N° Cluster*)	**Tempo (s)**	**Risultato**
Nearest Neighbor	N/A	526	1 812 602,19
Nearest Neighbor + prime swap	N/A	695	1 811 953,68
Concorde	N/A	77	1 524 920,15
Concorde + prime swap	N/A	200	1 524 915,60
Cluster 1: Concorde + Concorde	2000	1 772	1 574 601,37
Cluster 2: TSP Solver + prime swap	39	5 220	1 603 509,21

5 Discussione

L'obiettivo del progetto è stato trovare approcci che potessero migliorare il risultato ottenuto dal *Nearest Neighbor*. Fissato esso come principio guida, si sono considerati anche altri due aspetti che hanno determinato la scelta di alcuni approcci rispetto ad altri. Il primo riguarda l'uso delle risorse delle macchine virtuali (in quanto limitate), il secondo aspetto invece è relativo all'efficacia del metodo, ovvero la capacità di ottenere risultati paragonabili a quelli presenti nella Leaderbord di Kaggle.
L'approccio standard *Nearest Neighbor* impiega un tempo ridotto per ottenere una soluzione accettabile, ma sicuramente migliorabile. Come si è visto nella sezione 3.5, tale miglioramento si può effettuare applicando meccanismi di *fine-tuning* oppure modificando l'algoritmo (come sviluppi futuri) in modo tale che il calcolo del percorso tenga conto, in fase di creazione della sequenza, anche delle penalità che si possono verificare. Tutte le tecniche ideate ed implementate hanno permesso di ottenere dei risultati in linea con gli obiettivi di progetto, in particolare la soluzione ottenuta applicando *Concorde* in combinazione con *Prime Swap*, è risultata essere la migliore sia dal punto di vista dell'efficacia (distanza ricavata) che efficienza (tempo impiegato). Bisogna evidenziare però che la tecnica di ottimizzazione produce un miglioramento esiguo rispetto ad utilizzare unicamente *Concorde* sull'intero dataset, quindi la maggior parte dei meriti si possono attribuire al solver che è risaputo essere uno dei migliori per risolvere il problema TSP.
Analizzando i metodi alternativi (Cluster 1, Cluster 2) che sfruttano la tecnica di clustering, si osservano due aspetti interessanti. Il primo è che il k

ottimale emerso dal processo di *AutoML* risulta essere differente per i due approcci: nel primo caso k assume un numero elevato (2000) mentre nell'altro k è rappresentato da un valore piccolo (39). La seconda cosa interessante che emerge è la disparità dei tempi di esecuzione: si può osservare come il secondo metodo impieghi quasi il triplo del tempo per produrre la soluzione rispetto al primo. Il motivo di tale situazione può essere legata al fatto che il secondo approccio, per decretare l'ordine di visita dei cluster, deve calcolare ad ogni step una matrice di distanza in cui le righe rappresentano i punti estremi del gruppo che si sta considerando mentre le colonne i punti cardinali di tutti gli altri cluster non ancora visitati. Quindi tale operazione risulta essere molto più dispendiosa rispetto a quella prevista dal primo metodo, ovvero calcolare i centroidi e poi in base a quelli decretare l'ordine di visita dei cluster. Confrontando i due approcci quindi si evince che il migliore, in questo contesto applicativo, risulta essere il primo (Cluster 1) sia per il tempo di esecuzione che per la distanza complessiva restituita.
Si sottolinea infine che i due metodi differenti prevedono l'utilizzo di due diversi solver (*Concorde* per il primo e *TSP Solver* per il secondo), ciascuno risulta essere il migliore rispetto al contesto di utilizzo.

6 Conclusioni

Nel report sono state presentate diverse metodologie per la risoluzione del problema TSP asimmetrico, con particolare attenzione verso i limiti imposti dalle risorse disponibili. Gli esiti sono ritenuti soddisfacenti in quanto ogni approccio applicato produce una soluzione migliore rispetto al *Nearest Neighbor*; si osserva inoltre che i risultati ottenuti (esclusa la soluzione ricavata dal *Nearest Neighbor*) sono tutti paragonabili alle prime 1200 soluzioni migliori registrate nella leaderboard Kaggle. Questo però non implica che non si possa apporre dei miglioramenti futuri alle metodologie presentate. Tra tutti i possibili sviluppi si decide di riportarne due in particolare, con lo scopo di affinare il lavoro svolto. Il primo prevede di modificare per ogni solver la funzione di distanza, in modo tale che tenga conto dinamicamente del vincolo di penalizzazione (con questa modifica si potrebbe verificare se risulti più efficiente tenere conto della penalità in fase di creazione della sequenza oppure considerarla in un secondo momento). L'altro sviluppo riguarda la possibilità di affinare gli approcci "cluster-based", ottimizzando il meccanismo di unione delle soluzioni parziali ottenute per ogni cluster.

References

[1] Wikipedia, "Problema del commesso viaggiatore." [Online]. Available: https://it.wikipedia.org/wiki/Problema_del_commesso_viaggiatore

[2] G. Gutin, A. Yeo, and A. Zverovich, "Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the tsp," *Discrete Applied Mathematics*, vol. 117, no. 1, pp. 81 – 86, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0166218X01001950

[3] D. Applegate, W. Cook, and A. Rohe, "Chained lin-kernighan for large traveling salesman problems," *INFORMS Journal on Computing*, vol. 15, no. 1, pp. 82–92, 2003.

[4] M. Padberg and G. Rinaldi, "A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems," *SIAM review*, vol. 33, no. 1, pp. 60–100, 1991.

[5] Dmitry, "tsp-solver," https://github.com/dmishin/tsp-solver, 2018.

[6] M. Omran, A. Engelbrecht, and A. Salman, "An overview of clustering methods," *Intell. Data Anal.*, vol. 11, pp. 583–605, 11 2007.

[7] Z. Weng, "From conventional machine learning to automl," *Journal of Physics: Conference Series*, vol. 1207, p. 012015, 04 2019.

[8] G. A. Croes, "A method for solving traveling-salesman problems," *Operations research*, vol. 6, no. 6, pp. 791–812, 1958.

7 Appendice

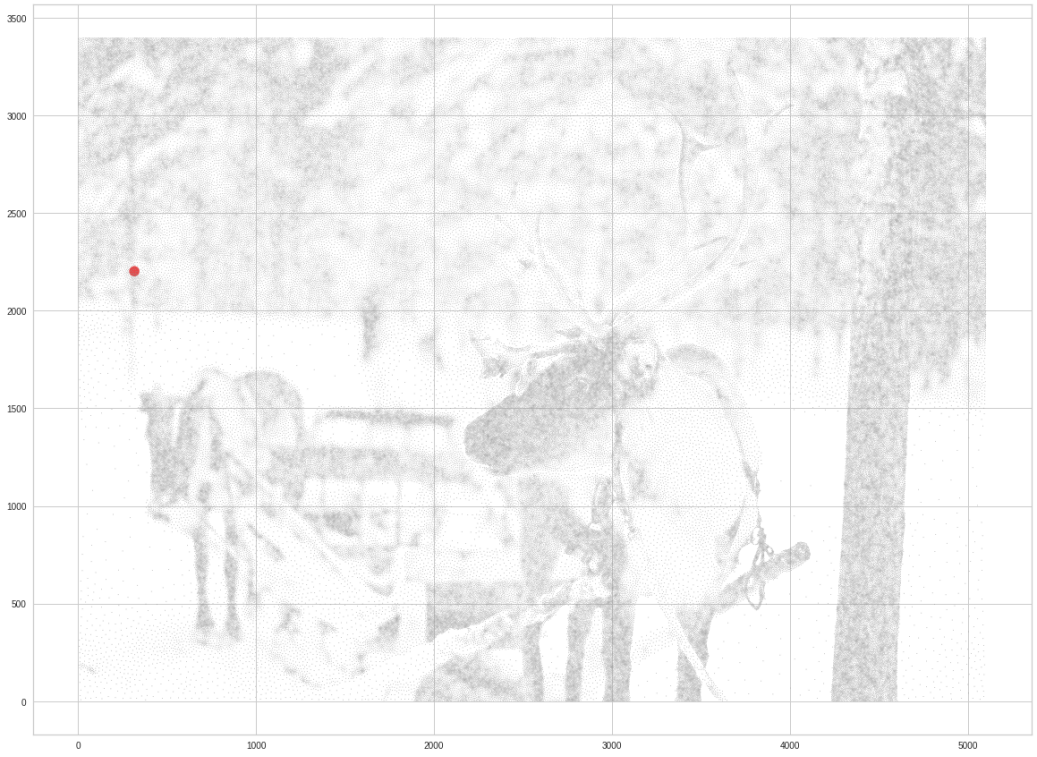

Figure 1: Mappa delle città contenute nel dataset (punto rosso rappresenta il Polo Nord)

Algorithm 4 Cluster primo metodo

procedure FINAL SOLVER(elenco città I con relative distanze, k numero di cluster)
    $clusters \leftarrow Gaussian\ Mixture(k,\ I)$
    for each c in $clusters$ **do**:
        Per ogni cluster c si calcolano i relativi centroidi
    $ordine \leftarrow NearestNeighbor(centroidi)$ Si applica *Nearest Neighbor* (o in alternativa *Concorde TSP*) sui centroidi dei cluster per determinare l'ordine della visita dei cluster
    $sequenza\_città \leftarrow [\ ]$
    $Cluster\_Polo\_Nord \leftarrow$ cluster contenente Polo Nord
    $sequenza\_città \leftarrow [NearestNeighbor(ClusterPoloNord)]$ Restituisce la sequenza di visita delle città contenute nel cluster Polo Nord
    for each $centr$ in $ordine$ **do**:
        $cluster \leftarrow$ cluster a cui è associato il centroide $centr$
        $cluster\_prec \leftarrow$ cluster precedente a $cluster$ nella sequenza $ordine$
        $dist \leftarrow$ punto di collegamento con distanza minima tra $cluster$ e $cluster\_prec$
        $temp \leftarrow NearestNeighbor(cluster)$
        Riordinamento della sequenza: si effettua per far si che il primo elemento della sequenza sia quello che determina la distanza minima tra i due cluster in maniera combinata con $dist$.
        $sequenza\_città \leftarrow sequenza\_città[dist :] + temp + sequenza\_città[: dist]$ Restituisce l'ordine di vista delle città appartenenti cluster considerati
    $sequenza\_città \leftarrow PrimeSwap(sequenza\_città)$
    $distanza\_totale \leftarrow total\_distance\_v2(sequenza\_città)$
    return $\frac{1}{distanza\_totale}$

Algorithm 5 Cluster secondo metodo: parte 1

procedure COMPUTE_DISTANCE(elenco città I con relative coordinate X e Y, k numero di cluster)
    cluster ← *Gaussian Mixture(k, I)*
    estremi_cluster ← Individuazione punti estremi per ogni cluster
    cluster_candidati ← *cluster* lista di cluster da visitare
    cluster_analisi ← *len(cluster_candidati)*
    distanza ← *tourTSP(cluster_candidati,cluster_analisi,estremi_cluster)* ritorna distanza del percorso di vista di tutti i cluster
    return *distanza*

procedure TOURTSP(elenco di *cluster da visitare*, *numero di cluster da visitare*, elenco di *punti estremi di ogni cluster*)
    ordineCluster ← [ ]
    linkPunti ← [ ]
    utilizzati ← [ ]
    for each *i* in *cluster_analisi*-1 **do**:
        Per ogni cluster *i* (partendo da quello che contiene Polo Nord) si estraggono gli estremi del cluster *i* e quelli di tutti gli altri cluster (*elementi_cluster_altri*) contenuti in *cluster_candidati*, escludendo quelli contenuti in *utilizzati* (quelli già visitati).
        inserimento *i* (Polo Nord) in *ordineCluster* (solo la prima volta)
        rimozione *i* (Polo Nord) in *cluster_candidati* (solo la prima volta)
        creazione di *MatriceDistanza(elementi_cluster,elementi_cluster_altri)*, che ritorna il collegamento (*punti*) e il prossimo cluster da visitare (*prossimo*)
        inserimento *prossimo* in *ordineCluster*
        rimozione *prossimo* in *cluster_candidati*
        inserimento *punti* in *linkPunti*
        inserimento *punti* in *utilizzati*
        i ← *prossimo*
    tsp_cluster ← *mergeTour(ordineCluster,linkPunti)* ritorna percorso TSP ottimo per ogni cluster
    path_cluster ← concatenazione di tutti i percorsi tsp ottimi per ogni cluster (*tsp_cluster*
    path ← *prime_swap*(*path_cluster*) applicazione Prime Swap su percorso
    distance ← calcolo distanza su *path*
    return $\frac{1}{distance}$

Algorithm 6 Cluster secondo metodo: parte 2

procedure MERGETOUR(*ordineCluster*,*linkPunti*)
    $a \leftarrow 0$ indice per scorrere la lista dei link
    for each i in *ordineCluster* **do**:
        Estrazione punti del cluster i
        Creazione matrice di distanza con punti cluster $i \leftarrow matrix$
        Se i contiene Polo Nord *startIndex* $\leftarrow$ *0* altrimenti *startIndex* $\leftarrow$ *linkPunti[a][0]*
        Se i è ultimo cluster *finishIndex* $\leftarrow$ *0* altrimenti *finishIndex* $\leftarrow$ *linkPunti[a][1]*
        $a \leftarrow a + 1$
        $path \leftarrow solve\_tsp(matrix, startIndex, finishIndex)$ applicazione algoritmo TSP Solver che ritorna cammino TSP per cluster i
        Salvataggio path cluster i
    return tutti i path