UNIVERSITÀ DEGLI STUDI DI CATANIA
DIPARTIMENTO DI MATEMATICA E INFORMATICA
CORSO DI LAUREA MAGISTRALE IN INFORMATICA

CAPACITED VEHICLES ROUTING PROBLEM WITH A HYBRID IMMUNOLOGICAL ALGORITHM

PROGETTO COMPUTAZIONE NATURALE

Antonio Ganci
Matricola: W82000167

ANNO ACCADEMICO 2019/2020

Indice

1 Introduzione

Qualsiasi attività di trasporto di beni e/o servizi da un punto di raccolta a una o più destinazioni può essere considerato un problema di distribuzione.
Il Vehicle Routing Problem (VRP) [4] è uno dei classici problemi di ottimizzazione nell'ambito della ricerca operativa che consiste nella gestione di un insieme di veicoli che, partendo da un deposito comune o *depot*, devono essere distribuiti su un certo numero di città al fine di determinare una serie di percorsi di costo minimo per ciascun veicolo in modo da soddisfare un dato insieme di clienti. Esistono diverse varianti del VRP, tra le più studiate troviamo sicuramente il Capacitated Vehicle Routing Problem (CVRP) [3] dove ad ogni singolo veicolo è imposta una capacità limitata di carico delle merci che devono essere consegnate.
Il CVRP ha molte applicazioni nella vita reale, basti pensare a qualsiasi azienda che si trova a distribuire i propri prodotti dalla sede centrale cercando di minimizzare il costo del trasporto pur soddisfacendo tutti i clienti, o ai servizi di raccolta rifiuti, o ancora al servizio di trasporto studenti. In questo progetto verrà affrontato il CVRP attraverso un algoritmo immunologico ibrido.

2 Definizione del problema

Il CVRP [1] è formalmente definito come un grafo non orientato $G = (V, E)$ dove $E = \{(v_i, v_j) | v_i, v_j \in V, i < j\}$ è un insieme di archi e $V = \{v_0, v_1, ..., v_n\}$ è un insieme di vertici in cui:

- v_0 corrisponde al *depot* o deposito;
- $V - \{v_0\}$ l'insieme delle città.

Ad ogni arco (i, j), con $i \neq j$, viene associato un valore non negativo d_{ij}, detto *travel cost* o *travel time*. Sia Q l'insieme delle richieste dei clienti da soddisfare $(q_i, \forall i \in V - \{v_0\})$, ed m il numero dei veicoli a disposizione ai quali è fissata una capacità C di merce trasportabile.
L'obbiettivo del CVRP è quello di determinare un insieme di *k-routes* di costo minimo, tale che vengano soddisfatte le seguenti condizioni:

1. ogni percorso inizia e finisce al *depot*;
2. $\sum_{k=1}^{m}(\sum_{i=1}^{|V_{route_k}|} q_i \leq C_k)$, dove V_{route_k} rappresenta l'insieme delle città appartenenti all'itinerario ($route_k$) del veicolo k;
3. ogni città in $V - \{v_0\}$ è visitata una sola volta da un solo veicolo.

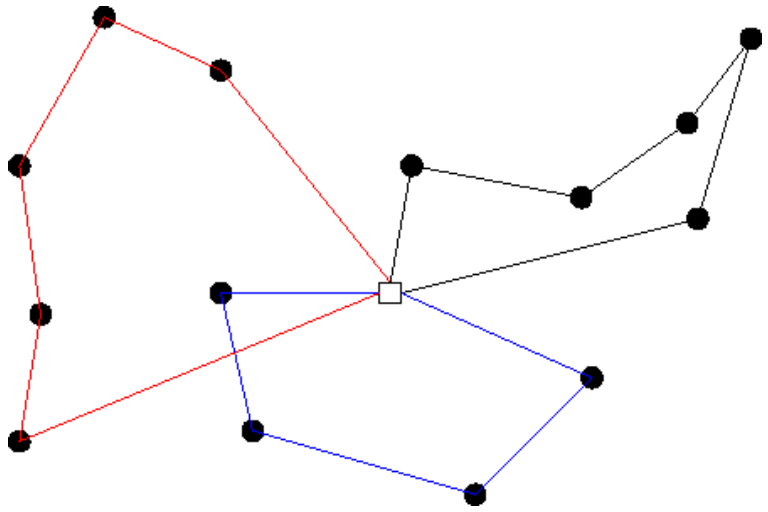

Figura 1: Esempio di Vehicle Routing Problem

3 L'algoritmo

Un algoritmo immunologico è un algoritmo bio-ispirato basato sul clonal selection principle. L'idea alla base di questo algoritmo è cercare di emulare il comportamento del sistema immunitario naturale, in particolare come le cellule si adattano per legarsi e quindi eliminare entità sconosciute meglio conosciute come antigeni. L'algoritmo si basa su due elementi principali, gli **antigeni** appunto, che rappresentano il problema da affrontare e le **B-cells** (Linfocita B) che rappresentano un punto nello spazio di ricerca [2]. L'algoritmo crea una popolazione di B-cell che cresceranno e si perfezioneranno nel corso delle generazioni fino a raggiungere un'età prefissata in cui andranno incontro alla morte. La qualità di ogni B-cell è valutata attraverso una funzione di fitness.
L'algoritmo parte generando una popolazione casuale $P^{(t)}$, dopodiché, ad ogni epoca, applica i seguenti operatori:

1. **Cloning.** Si tratta di un operatore che genera una popolazione intermedia $P^{(clo)}$ copiando *dup* volte la popolazione iniziale $P^{(t)}$, assegnando ad ogni nuovo elemento della popolazione un'età casuale tra $[0, \frac{2}{3}\tau_b]$, dove τ_b rappresenta l'età massima che un elemento della popolazione può avere. La nuova popolazione avrà dunque dimensione $d \times dup$.

2. **Inversely Hypermutation.** Questo operatore si occupa di esplorare il vicinato di ogni clone creato dall'operatore di cloning. Ciò viene eseguito mutando ogni elemento di $P^{(clo)}$ M volte, ottenendo così la nuova popolazione $P^{(hyp)}$. Per determinare il numero di mutazione M, viene adottata una legge inversamente proporzionale al valore della fitness. In generale, più piccolo è il valore della funzione di fitness, più la B-cell è buona (problema di minimizzazione) e quindi meno mutazioni saranno eseguite a quest'ultima. Dato una B-cell x, il valore di M è dato da:

$$M = \lfloor (a \times l) + 1 \rfloor \tag{1}$$

   con l lunghezza della B-cell.
   Il termine α rappresenta il mutation rate ed è dato da:

$$\alpha = e^{-\rho \hat{f}(x)} \tag{2}$$

   dove ρ è lo shape del mutation rate e $\hat{f}(x)$ è il valore della fitness normalizzato tra $[0, 1]$. Una mutazione consiste nell'applicazione di un operatore di perturbazione della soluzione. L'algoritmo sceglie con una probabilità del 50% tra l'operatore **move** e l'operatore **reverse**. Il primo sceglie un nodo casuale e lo inserisce in una posizione, anch'essa casuale, shiftando tutto il resto. Il secondo sceglie un intervallo casuale e inverte l'ordine dei nodi all'interno di esso.

3. **Aging.** Si tratta di un operatore che impedisce all'algoritmo di restare bloccato in qualche ottimo locale. Quello che fa è rimuovere tutte quelle B-cell la cui età (che viene incrementata ad ogni generazione) è

superiore all'età massima consentita τ_b. In questo modo si previene la convergenza prematura dell'algoritmo in qualche ottimo locale. Comunemente, viene utilizzata una versione dell'operatore di aging che prevede l'uso dell'*elitarismo*. In poche parole, la B-cell migliore viene comunque mantenuta a prescindere dalla sua età.

4. $(\mu + \lambda)$**-Selection.** L'operatore di selezione si occupa di scegliere le B-cell che verranno mandate alla generazione successiva. Le migliori d B-cell vengono selezionate tra la popolazione iniziale di grandezza $\mu = d$ e la popolazione hypermutata di dimensione $\lambda = (d \times dup)$. A volte può accadere che il numero di B-cell sopravvissute è minore di d. In questo caso, vengono generate in modo casuale delle nuove B-cell per riempire la popolazione da mandare alla nuova generazione. In generale, è possibile adottare criteri di selezione differenti. In questa implementazione si è scelto di mandare alla generazione successiva le migliori $0.6 \times d$ B-cell e le peggiori $0.4 \times d$. Ciò significa che, se la popolazione iniziale d è pari a 100, allora verranno selezionate le prime 60 B-cell (soluzioni con fitness più bassa, quindi migliore) e le ultime 40 B-cell (soluzioni con fitness più alta, quindi peggiore) dalla popolazione hypermutata λ. Mantenere anche una percentuale di B-cell "scarse" crea maggiore eterogeneità tra la popolazione e quindi maggiore possibilità di evitare gli ottimi locali.

5. **Local Search.** In questa versione dell'algoritmo è stato aggiunto un operatore di local search. Questo per cercare di raffinare ancora di più le B-cell selezionate in precedenza dall'operatore $(\mu+\lambda)$-Selection. L'algoritmo scelto è il *two-opt* la cui idea di base è quella di cercare un percorso in cui due o più archi si incrociano e riordinarlo in modo che questo non accada più.

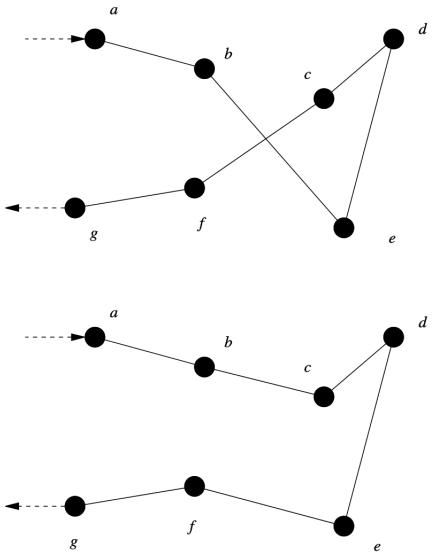

Figura 2: Esempio di 2-opt

Nel caso del CVRP, il *two-opt* viene applicato su ogni route della soluzione. L'idea è quella di manipolare la route fin quando questa non può più essere migliorata. Ad ogni route viene applicata la procedura mostrata nel listato 1:

Listing 1: two-opt swap.

```
for i in range(route_len):
    for j in range(i+1, route_len):
        # Swap elements with 2-OPT swap
        new_route = []
        for k in range(0, i):
            new_route.append(route[k])
        for k in range(j, i-1, -1):
            new_route.append(route[k])
        for k in range(j+1, route_len):
            new_route.append(route[k])
```

Una ricerca completa con *two-opt* esplorerà tutte le possibili combinazioni valide per ogni route. Ciò garantisce un tempo di convergenza, in termini

di numero di epoche, relativamente basso rispetto alla versione dell'algoritmo senza local search. Ovviamente, se da un lato diminuisce il tempo di convergenza, dall'altra aumenta il tempo di esecuzione di ogni singola epoca. Per istanze piccole, il problema non si pone, poiché il minor numero di epoche necessarie per la convergenza compensa il tempo di esecuzione di ogni epoca. Ma per istanze molto grandi non è più così. Si è sentita quindi l'esigenza di creare un compromesso tra numero di epoche necessarie alla convergenza e tempo di esecuzione di ogni epoca. Si è visto che eseguendo la local search dopo la selezione della nuova popolazione anziché prima, quindi eseguendola sulla popolazione selezionata per la nuova generazione e non su quella hypermutata si ha un risparmio in termini di tempo di esecuzione di ogni singola epoca (questo perché la popolazione selezionata è minore di quella hypermutata di un fattore *dup*) ma si mantengono in media lo stesso numero di epoche necessarie per la convergenza dell'algoritmo.

4 Implementazione

L'implementazione dell'algoritmo immunologico è stata effettuata in Python. Il codice si compone di quattro classi principali:

- **Instance.** Contiene tutte le informazioni essenziali alla risoluzione del problema (capacità, numero di città, numero di veicoli, ecc.). Inoltre, si occupa di calcolare la matrice delle distanze che verrà usata nei passi successivi dell'algoritmo.
- **Node.** Rappresenta ogni singolo cliente/città da visitare. Ogni nodo contiene le sue coordinate, l'id che lo rappresenta e la richiesta da soddisfare.
- **Solution.** Definisce la struttura della B-cell e i metodi applicati ad essa. Una B-cell o soluzione è rappresentata come un array di interi rappresentanti gli id dei nodi (clienti). L'array contiene $k + 1$ nodi *depot*, dove k è il numero di veicoli dell'istanza. I nodi giacenti tra un *depot* e un altro rappresentano una route o percorso.

0	5	3	8	0	1	2	0	9	7	4	0

Tabella 1: Esempio di B-cell con 10 nodi e 3 veicoli.

  Ogni B-cell non viene inizializzata casualmente ma seguendo la seguente procedura:

  1. Si sceglie un nodo (cliente) casuale dalla lista nodi;
  2. Si inserisce il nodo nella route che minimizza il costo;

  Al momento dell'inizializzazione, inoltre, alla B-cell viene assegnata un'età casuale compresa tra $[0, \frac{2}{3}\tau_b]$ e una *fitness* calcolata sommando le distanze tra i nodi di ogni route. Al momento del calcolo della fitness può accadere che il peso di una singola route ecceda la capacità totale del veicolo, in questo caso la soluzione viene marcata come *not feasible* e alla fitness totale viene aggiunta una penalità proporzionale alla violazione della B-cell. La violazione è data dalla somma delle quantità in eccesso su ogni route della soluzione mentre la penalità non è altro che la violazione moltiplicata per un fattore di scala pari a 100.
- **Population.** Gestisce la popolazione di B-cell epoca per epoca. La classe Population mantiene due array, *solutions* e *cloned_solutions*. Il primo contiene le soluzioni che epoca dopo epoca vengono mandate avanti mentre il secondo contiene i cloni delle soluzioni su cui, successivamente, vengono effettuate le mutazioni. Population mantiene sempre un riferimento alla soluzione migliore e a quella peggiore per ogni epoca.

L'intero algoritmo viene definito e inizializzato nel file `main.py`. Qui è possibile settare tutti i vari parametri dell'algoritmo, il numero di epoche e di runs da effettuare e su quale istanza effettuarle.

5 Risultati

Per il testing dell'algoritmo si è utilizzato un set di istanze standard[1] da cui se ne sono selezionate cinque differenti, di complessità crescente. Le istanze in questione sono:

- A-n32-k5: 32 clienti e 5 veicoli;
- A-n39-k6: 39 clienti e 6 veicoli;
- A-n45-k7: 45 clienti e 7 veicoli;
- A-n62-k8: 62 clienti e 8 veicoli;
- A-n80-k10: 80 clienti e 10 veicoli.

Per ogni istanza si sono realizzate 5 runs di 10000 epoche ciascuna e sono state registrate *best solution*, *worst solution*, *mean* e *standard deviation*. Il tetto massimo di 10000 epoche è sembrata una scelta ragionevole in quanto si è notato che l'algoritmo tende a "fermarsi" sempre abbondantemente prima delle 10000 epoche o perché ha trovato la soluzione ottimale o perché è rimasto bloccato in un ottimo locale. Per ogni run si è mantenuta una popolazione di 100 elementi con un'età massima di 5 epoche, un fattore di duplicazione (*dup*) pari a 2 e un mutation rate (ρ) di 10. La scelta di questi parametri è stata dettata sicuramente da una grande quantità di test effettuati prima dei test definitivi. In particolare si è visto che con un'età massima minore di 5 l'algoritmo sembrava "rimbalzare" tra una soluzione e l'altra senza mai avvicinarsi troppo a quella ottimale, mentre con un'età massima maggiore di 5 la probabilità di incappare in un ottimo locale era più alta. Per quanto riguarda *dup* si è invece visto che i risultati migliori si avevano con un fattore di duplicazione piccolo e una popolazione abbastanza grande. Anche la scelta di ρ è stata effettuata a seguito di svariati test, in particolare si trattava di scegliere indirettamente quanto mutare ogni soluzione e cercare il giusto compromesso tra stravolgere completamete la soluzione e non cambiarla affato. I risultati dell'algoritmo su ogni istanza sono mostrati nella Tabella 2.

	Best	**Worst**	**Mean**	**Std. Dev.**	**Best Known**
A-n32-k5	**784**	784	784	0	**784**
A-n39-k6	**831**	847	840	7.15	**831**
A-n45-k7	**1146**	1211	1165.2	26.3	**1146**
A-n62-k8	1303	1321	1313.6	8.23	1290
A-n80-k10	1817	1855	1842	16	1764

Tabella 2: Risultati dell'algoritmo su ogni istanza.

Come si può vedere dalla Tabella 2, in generale, più l'istanza diventa complessa più ci si allontana dalla soluzione ottimale. Si può infatti notare che per

[1] https://neo.lcc.uma.es/vrp/vrp-instances/capacitated-vrp-instances/

le prime tre istanze la soluzione ottimale è stata trovata e addirittura per la *A-n32-k5* è stata trovata in ognuna delle 5 runs, mentre per le istanze *A-n62-k8* e *A-n80-k10* pur ottenendo una buona soluzione, l'algoritmo non ha trovato quella ottimale. La complessità dell'istanza del problema e quindi del landscape del problema, fa si che l'algoritmo rimanga bloccato in un ottimo locale. Una possibile causa del problema potrebbe essere il fatto che l'algoritmo, avvicinandosi tantissimo alla soluzione ottima, e quindi a soluzioni con una fitness relativamente bassa, non riesce ad effettuare un numero di mutazioni tali da stravolgere la miglior soluzione trovata fino a quel momento, in pratica non riesce a spostarsi molto dal vicinato della soluzione migliore. Questo potrebbe essere un "effetto collaterale" dell'applicazione della local search dopo la selezione della generazione successiva. Un'altra possibile causa potrebbe anche essere banalmente un tempo di convergenza troppo elevato che non permette all'algoritmo di trovare la soluzione ottima in sole 10000 epoche. In questo caso, il porting del codice in un linguaggio più efficiente e la parallelizzazione dello stesso, che miglioreranno di parecchio le prestazioni computazionali, permetteranno di incrementare il numero massimo di epoche per ogni run dell'algoritmo.

6 Conclusioni

In questo progetto è stato affrontato il Capacitated Vehicle Routing Problem (CVRP) con un algoritmo immunologico a cui è stato aggiunto l'operatore di local search *two-opt*. I risultati mostrano che l'efficacia dell'algoritmo varia in base alla complessità dell'istanza del problema. In particolare, l'algoritmo riesce a trovare abbastanza facilmente la soluzione migliore in istanze del problema relativamente piccole mentre non riesce a farlo nelle istanze più grandi pur avvicinandosi parecchio. Potrebbe essere interessante testare l'algoritmo utilizzando diversi operatori di hypermutation, magari creandone qualcuno più raffinato combinando quelli già esistenti. L'algoritmo può essere indubbiamente migliorato soprattutto per quanto riguarda le prestazioni computazionali magari cercando di parallelizzare il codice dove possibile e/o utilizzare un linguaggio di programmazione più efficiente ($C++$).

Riferimenti bibliografici

[1] Habibeh Nazif and Lai Soon Lee. Optimised crossover genetic algorithm for capacitated vehicle routing problem. *Applied Mathematical Modelling*, 36(5):2110–2117, 2012.

[2] Mario Pavone and Giuseppe Nicosia. Clonal selection - an immunological algorithm for global optimization over continuous spaces. *Journal of Global Optimization*, 53:1–40, 08 2011.

[3] Ted K Ralphs, Leonid Kopman, William R Pulleyblank, and Leslie E Trotter. On the capacitated vehicle routing problem. *Mathematical programming*, 94(2-3):343–359, 2003.

[4] Paolo Toth and Daniele Vigo. *The vehicle routing problem*. SIAM, 2002.