Università degli Studi di
Napoli Federico II

Dipartimento di Ingegneria Elettrica e delle
Tecnologie dell'Informazione

Corso di Intelligenza Artificiale

Elaborato Esercitativo

Work Projects

Milo Saverio
Pommella Michele
Previtera Gabriele

Prof. Neri Filippo

Anno 2016-2017

Work Project 2

2.1 Esercizio 1: Evolutionary Optimization

2.1.1 Esperimento 1: Ottimizzazione

Attraverso gli algoritmi forniti siamo stati in grado di sperimentare le differenti tecniche di *ricerca locale* su diversi problemi. Esse si basano sul principio dei miglioramenti successivi: si cerca in primo luogo una soluzione,anche non ottima, ed in seguito ci si concentra sull'ottimizzazione. Va considerato che spesso si riesce a determinare solo un massimo locale, ottenendo risultati più o meno soddisfacenti a seconda dell'applicazione.

Random Searching

Questo algoritmo prevede semplicemente la determinazione casuale di un insieme di soluzioni e la comparazione dei relativi costi per l'identificazione di quella a costo minimo.
Abbiamo effettuato differenti esperimenti variando il numero di soluzioni estratte casualmente: ad un suo eccessivo aumento non si ottengono necessariamente risultati nettamente migliori da giustificare l'evidente incremento della complessità computazionale in termini temporali; d'altro canto, un numero esiguo di soluzioni produce risultati dalla qualità incostante.

Hill Climbing

L'`Hill Climbing` inizialmente determina una soluzione ed i suoi "vicini", se uno di essi presenta un costo migliore della soluzione corrente,verrà scelta questa come l'attuale alla prossima iterazione, attuando, in tal modo, il processo dei miglioramenti successivi; se la soluzione corrente ha costo minore dei suoi vicini, termina la serie di miglioramenti iterativi poiché si è in presenza di un massimo locale (minimo locale rispetto al costo, in quanto una soluzione è preferibile da un altra se il costo è minore o maggiore dipende dal problema).

Vari esperimenti ci hanno condotto a risultati decisamente migliori del `Random Searching`, che sottolineano la maggior efficienza dell'`Hill Climbing`. La variabilità nella qualità della soluzione, al seguito di svariate esecuzioni, è dipendente dai massimi locali del problema specifico,infatti partendo da situazioni iniziali diversi, arriviamo a valori diversi.

Simulation annealing

Versione rivisitata dell' L'`Hill Climbing`, questo metodo di ottimizzazione è ispirato dal riscaldamento termico di una lega e al suo successivo raffreddamento. Viene comparata la soluzione corrente ad una determinata casualmente (nel nostro caso tramite un cambiamento di quella corrente): l'algoritmo tende a spostarsi sempre verso una soluzione successiva migliore, solo con una certa probabilità verso quelle peggiori, che tenderà a diminuire col tempo. Questa probabilità dipende dalla qualità della soluzione successiva: mosse pessime avranno minore probabilità. Viene, inoltre, introdotto un concetto di "temperatura": essa determina la probabilità di spostarsi verso soluzioni non ottime. Inizialmente questo valore sarà alto (temperatura elevata), in seguito essa tenderà a calare (raffreddamento), consentendo, in ultima battuta, la scelta della soluzione ottimizzata. Tale probabilità è data dalla seguente espressione:

$$e^{\frac{-(costosuccessivo+costoattuale)}{temperatura}}$$

Osserviamo che con l'abbassamento della temperatura, ad ogni iterazione dell' algoritmo, tale probabilità decresce. Le ragioni alla base di questa logica risiedono nella maggiore possibilità di trovare un massimo globale, evitando che ci si possa assestare su un punto di massimo locale come una *spalla* (shoulder).
Dalle prove sperimentali abbiamo constatato risultati migliori o uguali all'`Hill Climbing`, appurando la maggiore capacità di ottenere soluzioni con costo inferiore.

Genetic Algorithms

Ulteriore algoritmo che tenta di superare i limiti dell'`Hill Climbing` è quello genetico. Esso è ispirato dalla natura, proponendo un'evoluzione delle soluzioni tipicamente darwiniana. L'algoritmo prende le mosse selezionando un insieme casuale di soluzioni, detto *popolazione*. Gli esemplari di questa vengono poi valutati ed ordinati attraverso una *funzione di fitness* (rappresentata nel nostro caso dal costo minimo). Nell'implementazione utilizzata, si procede alla creazione della successiva *generazione* a partire da

un prestabilito numero di migliori soluzioni. Attraverso il processo di *elitarismo* quelle ottime sopravviveranno nella nuova generazione; la restante popolazione sarà costituita attraverso la *mutazione* (cambiamento casuale ad una soluzione) ed il *crossover* (combinazione di due soluzioni) dell'elite.
Vari esperimenti di hanno portato a riscontrare risultati simili al `Simulated Annealing`, se non migliori in taluni casi, che presentano, però, una minore variabilità con misure ripetute. Riusciamo, in definitiva, ad ottenere una buona ottimizzazione, di qualità superiore all'`Hill Climbing` ed al `Random Searching`.

2.1.2 Esperimento 2: Caso d'uso

L'ottimizzazione sulla quale ci vogliamo concentrare è quella di una funzione $F(x)$, così definita:

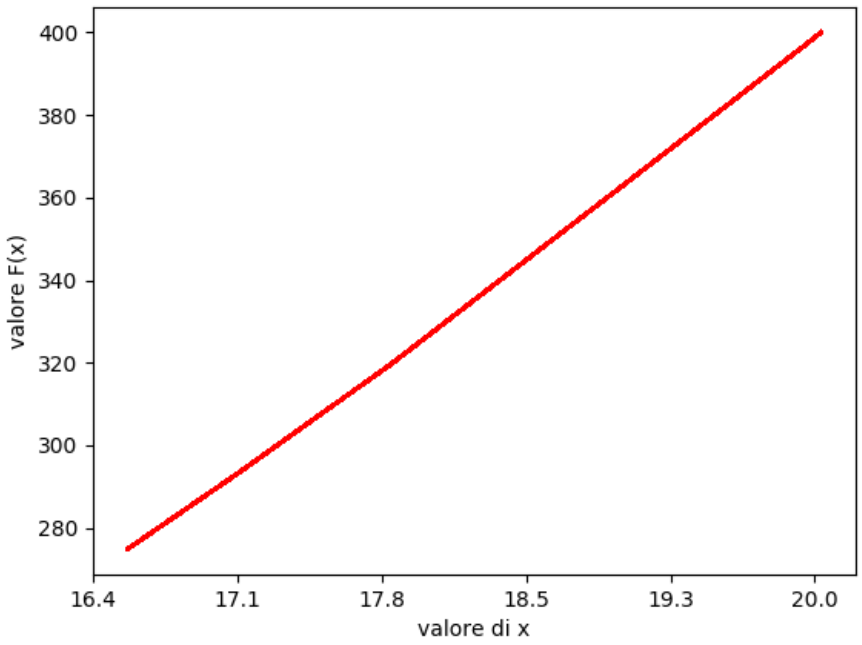
$$F(x) = \begin{cases} 10 & \text{se } x < 5.2 \\ x^2 & \text{se } 5.2 \leq x \leq 20 \\ x - 1 & \text{se } x > 20 \end{cases} ; x \in [-100, 100]$$

dove la x può assumere valori tra $[-100, 100]$. La ricerca di tale valore verrà effettuata tramite algoritmo genetico. Prima di tutto, però, dobbiamo definire la funzione di costo, così da poter determinare quando la nostra soluzione tende a migliorare o, viceversa, a peggiorare:

```
1 def costmax(sol):
2         if sol[0]>100 or sol[0]<-100:
3                 cost=0
4         else:
5                 cost=F(sol)
6 return cost
```

La nostra funzione di costo è definita nel seguente modo: se il valore di cui si calcola il costo non è accettabile, poiché nel passaggio tra una generazione ad un'altra si è avuto un esemplare della popolazione non voluto, cioè non accettabile per la soluzione (valore al di fuori dell' intervallo $[-100, 100]$), gli si dà un costo pari a 0, in modo tale che per la creazione della prossima generazione non verrà preso in considerazione; altrimenti il costo è dato dall'applicazione della F sul valore in ingresso.
L' algoritmo prima di tutto genera una popolazione iniziale scegliendo a caso valori presi nello spazio delle soluzioni, ne calcola i vari score e sceglie una

elite di individui (quelli che in questo caso massimizzano lo score), dopodiché, per un numero di generazioni volute, possiamo far mutare un elemento o fare il crossover tra due di essi in base ad una probabilità di mutazione da noi fornita.
Abbiamo però riscontrato un problema nella rappresentazione numerica: i dati vengono passati all' algoritmo sotto forma di float, che non permette di definire un' operazione di mutazione o crossover semplice. Per ovviare a questo inconveniente i numeri vengono convertiti nella loro forma binaria (in questo caso a 32 bit), definendo quindi la mutazione come il cambiamento di un bit da 1 a 0, o viceversa.
Il crossover è eseguito scegliendo un numero di bit casuale da una soluzione e il numero restante di bit da un'altra, affinché si riottenga un codice di 32 bit. Possiamo vedere il crossover come un'operazione di ricombinazione genetica in cui parte del materiale genetico di un individuo si ricombina con un altro. Così facendo, ad ogni cambio generazionale, si genera una progenie come combinazione di geni dei genitori migliori della generazione precedente, attuando in tal modo una selezione naturale.
La mutazione, invece, è un fattore casuale che permette di introdurre nuovi tratti negli individui, di variarne i geni, donando in tal modo alla progenie caratteristiche distintive non presenti nei genitori e consentendo alla specie di attuare un processo evolutivo che favorisca la sopravvivenza.



Come si può vedere, il valore che ottimizza la F(x) tende a x=20. Il risultato si riesce a trovare anche partendo da una popolazione iniziale ristretta a soli 100 elementi. Per un numero di iterazioni discreto (100 in questo caso) la soluzione non solo viene trovata, ma viene confermata più volte, dimostrando che, anche se si provano nuove combinazioni, l'esemplare che massimizza la fitness è stato già trovato precedentemente e le iterazioni successive non fanno altro che confermarlo.

2.2 Esercizio 2: MDS: Visually Exploring US Senator Similarity

Clustering

Il Clustering, o analisi di raggruppamento, è una tecnica di intelligenza artificiale volta alla selezione e raggruppamento di elementi omogenei in un insieme di dati. Tutte le tecniche di clustering si basano sul concetto di distanza tra due elementi, che ci permette di definire il concetto di somiglianza tra gli elementi. Infatti l'appartenenza o meno ad un insieme dipende da quanto l'elemento preso in esame sia distante dall'insieme. La bontà delle analisi ottenute dagli algoritmi di clustering dipende dalla metrica: metriche diverse porteranno quasi certamente a cluster differenti.

MDS: MultiDimensional Scaling

Il MultiDimensional Scaling è una tecnica di analisi statistica usata per mostrare graficamente le differenze o somiglianze tra elementi di un insieme. È una generalizzazione del concetto di ordinamento: partendo da un matrice contenente le informazioni che vogliamo analizzare e moltiplicandola per la sua trasposta, otterremo una matrice quadrata reppresentante la "somiglianza" di ogni elemento con un altro, dopodiché l'algoritmo di scaling multidimensionale assegnerà a ogni soggetto una posizione in uno spazio N-dimensionale, con N stabilito a priori tramite una metrica.

Riflessioni

L'esempio affrontato descrive l'orientamento politico dei membri dei Congressi americani, dal 101-esimo al 111-esimo, sull' analisi dei voti. Sono utilizzati i dataset in formato *STATA*, programma commerciale di calcolo statistico diffuso tra gli accademici nell'ambito delle scienze politiche. Essi presentano i voti delle differenti sedute dei 10 Congressi presi in esame. La metrica utilizzata per la distanza è quella euclidea. Essendo i dati troppo

complessi e strutturati per i nostri scopi, per facilitare l' analisi viene eseguita una semplificazione aggregando dei tipi di voto: tra tutte le possibili classi di votazione (nove in totale) se ne ricavano solo tre.
Viene dapprima analizzato il caso del 110-esimo Senato. Notiamo che gli schieramenti dei voti sono netti: Repubblicani a destra e Democratici a sinistra. Estendendo l'analisi a tutti i Congressi, constatiamo che le scelte di voto sono consistenti con il partito di appartenenza. Alcuni grafici possono sembrare meno polarizzati, caratterizzati da punti più centrali, come se i voti siano stati meno radicali, ma tale risultato si ottiene perché non si ha una scala univoca che definisca la distanza.

2.3 Esercizio 3: Constraint Satisfaction Problems

Il problema di soddisfacimento dei vincoli che ci ritroviamo ad affrontare è il seguente: "Un agente intelligente deve colorare le province della regione Campania evitando che le confinanti abbiano lo stesso colore". Questo è un tipico esempio di domanda a cui si può rispondere con algoritmi che tengono conto del soddisfacimento dei vincoli. Come prima cosa formalizziamo il problema :

Variabili: Caserta, Benevento, Napoli, Avellino, Salerno
Vincoli: province confinanti devono avere colori diversi

Possiamo, dunque, ricavare il grafo dei vincoli, i cui nodi sono le variabili ed i cui archi connettono coppie di attributi che partecipano ad un vincolo. In questo caso i vincoli sono binari, cioè sono coinvolti soltanto due nodi per ogni vincolo.

Analizziamo il caso in cui l'agente disponga di solo 2 colori.

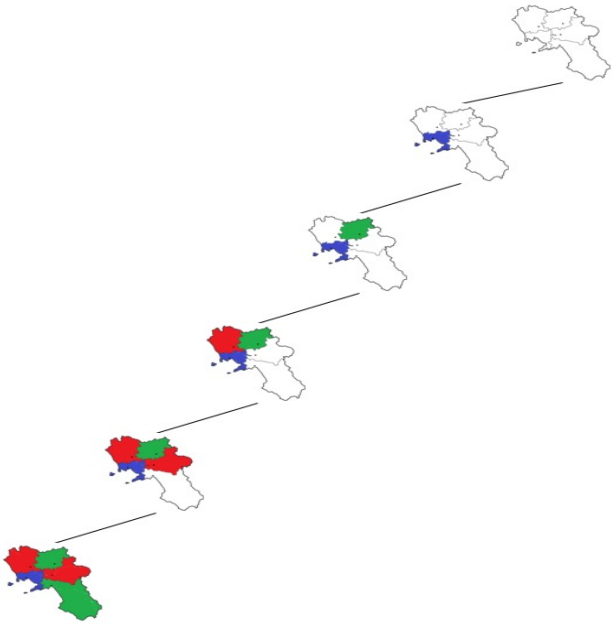
$$\text{Dominio} = \{\text{Colore1}, \text{Colore2}\}$$

Questo significa che le variabili possono assumere solo due colori. Da uno sguardo attento del grafo possiamo formalizzare un ulteriore vincolo, non binario: le variabili Napoli, Benevento e Avellino devono essere necessariamente tutte diverse (non sono l'unica tripla che rispetta questo vincolo). Un dominio formato da solo due colori non ci permetterà mai di poter soddisfare questo vincolo, pertanto il problema non è risolvibile. Ma oltre questa formulazione formale abbiamo provato anche l'algoritmo di `Backtracking Search`, attraverso il quale si perviene presto alla conclusione dell'impossibilità di risolvere il problema vincolato. Ad esempio, applicando le euristiche per migliorare l'efficienza della ricerca, assegniamo a Napoli (`Degree Heuristic`: variabile con più vincoli sulle rimanenti variabili) il valore Colore1; se procediamo assegnando Colore2 a Caserta (o qualsiasi altra variabile dato che ora presentano tutte un solo valore ammissibile), si ottiene un fallimento, poiché tramite il `Forward Checking` si osserva che Benevento non può assumere nessun valore ammissibile. Questa risultato è comune anche agli altri path dell'albero di ricerca e tutto ciò è intuitivamente comprensibile poiché con soli 2 colori non siamo in grado di soddisfare i loop di 3 nodi presenti nel grafo dei vincoli.

Analizziamo il caso in cui l'agente disponga di 3 colori.

$$\text{Dominio} = \{\text{Colore1}, \text{Colore2}, \text{Colore3}\}$$

Proviamo ora l'algoritmo con il dominio di dimensione maggiore. Partiamo da Napoli assegnandole il Colore1 e notiamo che questa scelta riduce i valori ammissibili per le altre province confinanti a due. Coloriamo, ora, Benevento con il Colore2; di conseguenza Caserta sarà la nostra prossima scelta poiché le si può assegnare solo un colore, il Colore3. Tale scelta è dettata dall'euristica che abbiamo applicato, il `Minimum Remaining Values`, che prevede la scelta della variabile con numero minimo di valori possibili. Ad Avellino, non confinante con Caserta, le possiamo assegnare il Colore3 senza produrre nessun conflitto. A Salerno, infine, assegniamo il Colore2. Osserviamo che ogni provincia ha un colore diverso dalle confinanti, ma queste ultime possono essere colorate allo stesso modo nel caso in cui non siano a loro volta adiacenti. Possiamo, in conclusione, affermare che è possibile soddisfare i vincoli dato questo dominio.



Anche il caso con quattro colori sicuramente permetterà di soddisfare i vincoli perché si è aumentato il numero di valori ammissibili lasciando invariate le informazioni iniziali. Verifichiamo che sia così.

$$\text{Dominio} = \{\text{Colore1}, \text{Colore2}, \text{Colore3}, \text{Colore4}\}$$

La prima provincia che coloriamo è Napoli, assegnandole il Colore1: così facendo riduciamo tutti i colori possibili per le altre province, avendo Napoli il maggior numero di vincoli. Avellino sarà la nostra prossima scelta: assegnandole il Colore2 diminuirà anche il numero di valori per Salerno e Benevento. Passiamo a Caserta: essa può assumere tutti i valori tranne Colore1. Osserviamo, però, che l'assegnazione dei valori Colore3 e Colore4 produrrebbe una diminuzione del dominio degli attributi che possiamo assegnare a Benevento. Applichiamo il `Least Constraining Value`, euristica che predilige il valore che lascia più libertà alle variabili adiacenti, e scegliamo il Colore2 per Caserta. Segue Benevento con il Colore3 e Salerno con il Colore3. Si noti che, per le ultime due scelte, si può effettuare una scelta completamente arbitraria tra Colore3 e Colore4.
Come ci si poteva aspettare, l'aggiunta di un nuovo elemento nel Dominio, senza l'alterazione dei vincoli, permette ugualmente la risoluzione del problema, perfino evitando di utilizzare il nuovo valore. Possiamo, allora, dedurre che, una volta trovata la soluzione con un determinato numero di valori, essa esisterà anche per un Dominio più esteso.
In quest'ultimo caso, però, si può parlare di vincoli più "elastici", più rilassati, grazie ai maggiori gradi di libertà per la determinazione della

soluzione. Ad esempio, con un Dominio di tre elementi il numero di soluzioni possibili sicuramente sarà molto più limitato rispetto ad uno con quattro elementi, che permette più opportunità di scelta. Possiamo, quindi, affermare che le soluzioni relative al Dominio esteso includano quelle del compatto.

2.4 Esercizio 4

Ricerchiamo un possibile approccio per trovare la soluzione con l' ausilio di algoritmi genetici al problema vincolato con $card(Dominio) = 4$. Prima di utilizzare gli algoritmi genetici per risolvere il problema, dobbiamo definire una funzione di costo. Tra le tanti possibili scelte, abbiamo optato per la seguente:

$$\text{costo} \;=\; \text{numero di conflitti}$$

dove il conflitto è inteso come due province confinanti con lo stesso colore. Una soluzione sarà quindi valutata rispetto ad una funzione di fitness che prende in considerazione il numero di conflitti: quanto più una soluzione presenterà un costo minore, tanto più assumerà un valore di fitness maggiore. Ovviamente ottimizzare la soluzione significa minimizzare il valore assunto dalla funzione di costo. Poiché essa può assumere solo valori positivi, la migliore soluzione riscontrabile avrà costo nullo.
Particolarizziamo l'algoritmo genetico per il problema in esame. La popolazione iniziale sarà costituita da un insieme di mappe che presentano una colorazione casuale delle province. Definiamo, poi, una probabilità di mutazione che consente di variare casualmente colore ad una provincia, tra i tre rimanenti (escludendo il colore attualmente assegnato). Tale operazione permette di aggiungere caratteristiche che la nostra popolazione iniziale non aveva, ottenendo maggiore varietà nel processo di generazione affinché si testino anche soluzioni non necessariamente eccellenti, infatti potrebbe capitare che tutti gli elementi della popolazione convergano verso una soluzione, avendo una convergenza verso i punti di massimo della funzione locali, invece che assoluti. Ora non ci resta che definire l'operazione di crossover, che consiste nel prendere in modo casuale dall'elite due elementi (mappe colorate) e scegliere probabilisticamente il numero di province di uno, da unire con le restanti dell'altroquesta operazioni serve per creare la nuova generazione, dopodiché il tutto si ripeterà ciclicamente fino a che non arriviamo alla soluzione ottimale descritta sopra o se il problema fosse troppo complesso accontentarci di una soluzione sub-ottima.