A solution for MWVC using Iterated Local Search

Laboratorio Intelligenza Artificiale (LM-18)
Università degli Studi di Catania - A.A 2021/2022

Danilo Leocata
Docente: Mario Pavone

March 20, 2022

1 Introduzione

Si propone una soluzione per il Weight Vertex Cover problem utilizzando l'Iterated Local Search: l'obbiettivo proposto è trovare la migliore soluzione, data un istanza, con il minimo numero di iterazioni. Il codice è disponibile al seguente repository di GitHub.
Il codice è stato implementato in Java utilizzando metodi custom. Inizialmente l'algoritmo è stato implementato con Python ma per via degli oneri computazionali (LPI richiedeva diverse ore per essere completato) è stato trovato più opportuno realizzare una versione in Java. Prima di procedere con l'implementazione e la scelta dell'algoritmo, sono state consultate e prese in considerazioni varie pubblicazioni riguardanti la soluzione del problema (indicate a fine relazione). È stata trovata sin da subito interessante implementare una soluzione utilizzando l'Iterated Local Search, in particolare se ne propone una versione che perserva soluzioni non ottime sfruttando un *term memory*. Sfruttando l'ILS, è possibile ottenere sin da subito una soluzione *completa* (intesa quando l'insieme dei nodi selezionati permette di raggiungere tutti i nodi dell'istanza) dopo poche iterazioni e migliorarla. Inoltre, sfruttando il *term-memory* è possibile tenere traccia delle migliori soluzioni trovate permettendo di accettare e continuare la ricerca su soluzioni peggiori senza perderne traccia avendo la possibilità di ripescarle nell'operatore di perturbazione.

2 Validità della soluzione

La soluzione è da considerarsi completa e valida se tutti gli archi del grafo sono esplorati. Di conseguenza è necessario controllare la validità della soluzione dopo la rimozione di un nodo. È stato implementato un modo per effettuare il controllo di validità in maniera efficiente dimezzando il costo computazionale considerando uguali archi invertiti (`(a,b) = (b,a)`) come uguali. È stata implementata una funzione per completare eventuali soluzioni non complete dalla `Perturbazione`.

Algorithm 1 `CompleteSolution`

```
Require: perturbed solution
  if
     perturbed solution is complete then
     do nothing, return perturbed solution
  else

     while
        perturbed solution is complete do
         Get list of unselected nodes
         find the best candidate node based on the number of knots (higher) or
  weight (lower) randomly and add to solution
     end while
     end if return CompletePerturbedSolution
```

Dai test effettuati è emerso che l'utilizzo dopo l'operatore di perturbazione tende a far bloccare la soluzione in un ottimo locale, di conseguenza verrà applicata solo dopo l'utilizzo della LocalSearch.

3 ILS

Sono stati testati sulle istanze due tipi di approcci:

- costruzione di una soluzione a partire da un nodo iniziale scelto randomicamente;
- scoperta della soluzione migliore partendo dalla soluzione peggiore (intesa così quella che ha selezionati tutti i nodi presenti nell'istanza);

I diversi approcci producono circa gli stessi risultati. È stato preso in considerazione, ma scartato durante la fase di test, il random restart della soluzione dato che non ha portato migliorie al risultato finale. Nel dettaglio, l'algoritmo ILS presentato parte dalla soluzione *peggiore*, con tutti i nodi dell'istanza selezionati, e per mezzo dell'operatore di perturbazione e della local search si cercherà di ottimizzare la soluzione, oppure da una soluzione completa costruita randomicamente

Algorithm 2 Iterated Local Search

```
Require: list of all nodes of instances
  CurrentSolution = Solution with all available nodes selected or Randomly while
  is complete
  initialize an empty term-memory
  while max evaluations do
     SolutionToChange = CurrentSolution
     apply Perturbation to SolutionToChange
     apply LocalSearch to SolutionToChange

     if cost of SolutionToChange < cost of CurrentSolution and SolutionToChange not
  in term-memory then
        add SolutionToChange in term-memory
     end if
        CurrentSolution = SelectCriteria(CurrentSolution, SolutionToChange,
  term-memory)
     end whilereturn CurrentSolution
```

4 Operatore di perturbazione

L'idea generale della perturbazione è quella di modificare i parametri ad ogni iterazione applicando una perturbazione sui nodi selezionati dalla soluzione. Un buon algoritmo deve evitare di far cadere sempre nello stesso minimo locale, in generale:

Algorithm 3 Perturbation

```
Require: Solution
  add random node from list of unselected if is possible
  remove random node from list of already selected
  return solution
```

Sono state implementate due versioni di perturbazione rispettivamente: `StrongPerturbation` che rimuove o aggiunge randomicamente nodi da 1 o al massimo `selectedNodes \ 2` e `WeakPerturbation` che rimuove/aggiunge un solo nodo dalla soluzione. Dai test effettuati portano circa agli stessi risultati. Inoltre è stato già dimostrato, da una delle referenze citate, che una perturbazione troppo forta porta allo stallo dell'ottimo locale.

Inoltre, tra le migliorie che la perturbazione potrebbe apportare alla soluzione vi è:

- l'eliminazione automaticamente di cicli se questa contiene dei nodi ridondanti;
- potrebbe rendere la soluzione non completa: di conseguenza applicando nuovamente la LocalSearch è possibile trovare un nodo candidato migliore rispetto a quello rimosso.

5 Local Search e criterio di selezione del nodo migliore

Algorithm 4 LocalSearch

Require: `Solution`
  if given solution has unreached nodes **then**
    `evaluate if swapping some nodes brings benefits to the solution`
    `increment counter for every evalutation`
  else
    `evaluate if removing some nodes brings benefits to the solution`
    `increment counter for every evalutation`
  end if
  `update solution`
  return updated solution

6 Criterio di accettazione

L'utilizzo del term memory permette di 'accettare' soluzioni peggiori rispetto ad altre e continuare la ricerca senza perder traccia di tutte le soluzioni trovate. In sintesi, se la soluzione perturbata è simile alla corrente il criterio di accettazione può ripescare, randomicamente una delle soluzioni presenti nel `term-memory` e continuare la ricerca. È stato introdotto inolte un `lockCounter` per evitare che la soluzione rimanga bloccata su un ottimo locale.

Algorithm 5 `Acceptance criteria`

```
Require: prev solution, new solution, term memory, lockCounter
  if
     cost old solution > cost new solution  return new solution then
  else
     if lockCounter = to maxValue
     return random solution from term-memory
end if
return random choice between (perturbed solution, current solution)
```

Tuttavia nella pratica questa ricerca non ha sempre portato a buoni risultati, di conseguenza è stato disabilitato.

7 Benchmarks e conclusione

È stato implementato uno script che permette di salvare i benchmark le soluzioni ottenute dalle istanze su file `.csv`, oltre alla creazione dei grafici di convergenza. I grafici importati ed le tabelle fanno riferimento alle run con `WeakPerturbation`. n certi casi, si rimane bloccati su un ottimo locale per molto tempo, fino a quando una soluzione perturbata non ottimale fa da punto di partenza per trovare una soluzione migliore. Si nota inoltre che un numero maggiore di nodi o archi non implica necessariamente un numero maggiore di valutazioni: anche grafi con lo stesso numero di nodi e archi hanno valori molto contrastanti, probabilmente perché il problema dipende molto dal peso dei nodi e dalla topologia del grafo.

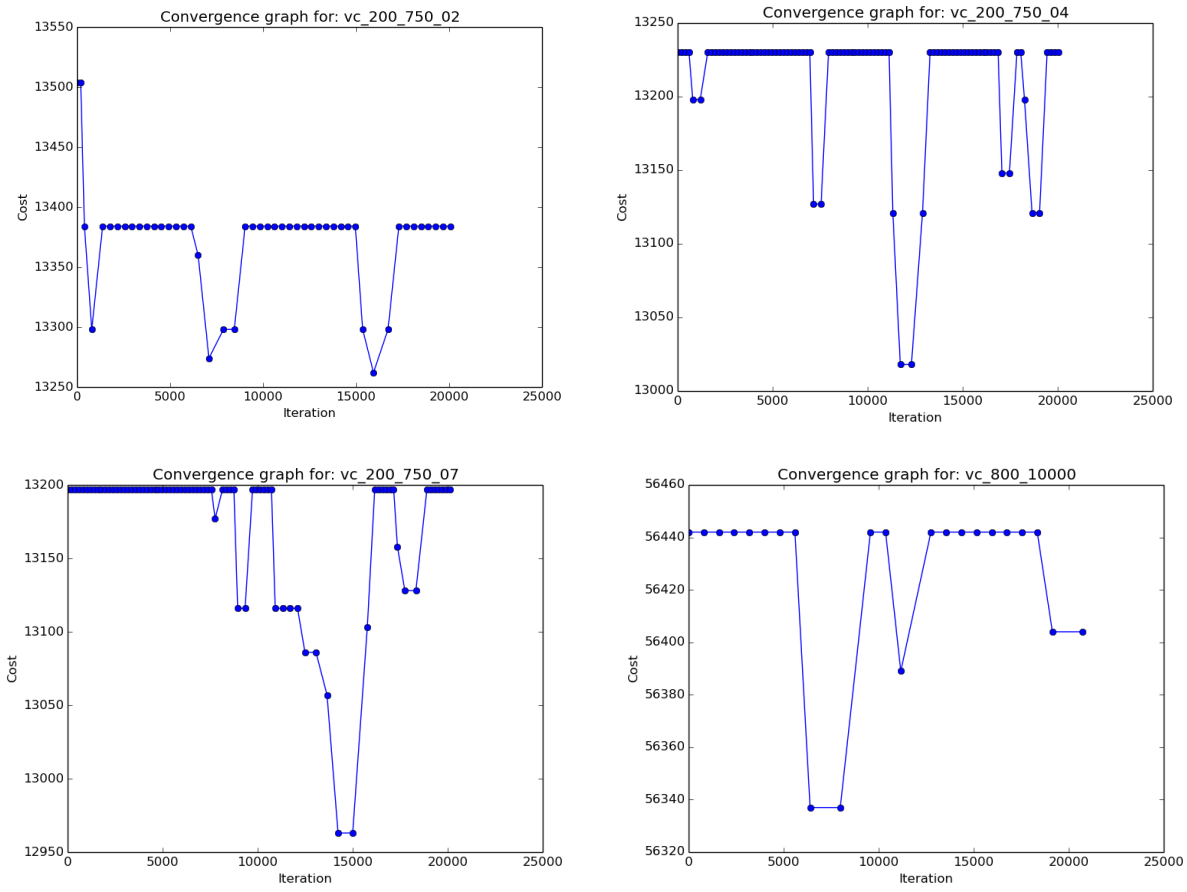

Figure 1: Sample grafici di convergenza

instance	best solution	best solution iter	elapsed ms
vc_20_120_01	1253	77	99
vc_20_120_02	1348	191	94
vc_20_120_03	1319	58	92
vc_20_120_04	1414	1179	99
vc_20_120_05	1293	761	92
vc_20_120_06	1271	77	102
vc_20_120_07	1265	1806	98
vc_20_120_08	1520	1654	98
vc_20_120_09	1587	229	97
vc_20_120_10	1452	381	95
vc_20_60_01	1105	405	33
vc_20_60_02	1585	476	37
vc_20_60_03	1313	58	34
vc_20_60_04	1177	115	37
vc_20_60_05	1352	324	37
vc_20_60_06	1426	343	33
vc_20_60_07	1573	210	36
vc_20_60_08	1526	39	36
vc_20_60_09	1595	324	36
vc_20_60_10	1416	552	35
vc_25_150_01	1785	265	108
vc_25_150_02	1472	241	103
vc_25_150_03	1658	20000	105
vc_25_150_04	1923	97	103
vc_25_150_05	1792	1849	114
vc_25_150_06	1651	505	99
vc_25_150_07	1977	841	104
vc_25_150_08	1693	361	107
vc_25_150_09	1739	841	111
vc_25_150_10	1547	553	102
vc_200_750_01	14160	1991	280
vc_200_750_02	13384	399	201
vc_200_750_03	14509	6767	297
vc_200_750_04	13230	797	306
vc_200_750_05	14495	399	211
vc_200_750_06	13586	1792	325
vc_200_750_07	13197	7762	286
vc_200_750_08	13676	9155	298
vc_200_750_09	13710	200	208
vc_200_750_10	13794	399	211

instance	best solution	best solution iter	elapsed ms
vc_100_2000_01	7027	1189	3158
vc_100_2000_02	6853	20000	2970
vc_100_2000_03	6680	20000	3032
vc_100_2000_04	6890	6535	2984
vc_100_2000_05	7236	8812	3093
vc_100_2000_06	6876	2080	2873
vc_100_2000_07	7322	4753	3103
vc_100_2000_08	7163	1090	2930
vc_100_2000_09	6829	793	3073
vc_100_2000_10	7238	1882	2977
vc_100_500_01	6739	5446	252
vc_100_500_02	7205	496	225
vc_100_500_03	6764	10331	220
vc_100_500_04	6818	9802	242
vc_100_500_05	7185	2278	254
vc_100_500_06	7228	3961	237
vc_100_500_07	6763	194	81
vc_100_500_08	6788	397	250
vc_100_500_09	6894	193	226
vc_100_500_10	6670	2773	244
vc_200_3000_01	13795	200	2648
vc_200_3000_02	13790	20000	4038
vc_200_3000_03	14245	8956	4144
vc_200_3000_04	15082	598	2681
vc_200_3000_05	14095	9354	4136
vc_200_3000_06	13731	2588	4196
vc_200_3000_07	14247	16717	4083
vc_200_3000_08	14197	200	3903
vc_200_3000_09	14294	19901	4141
vc_200_3000_10	13907	4976	3982
vc_800_10000	56404	6393	52037

Una delle difficoltà più grandi trovate durante l'implementazione è stato quello di determinare se i criteri di scelta fossero effettivamente robusti o no. Sarebbe interessante proseguire con lo studio del problema continuando sui seguenti punti:

- implementare altri criteri per la selezione del nodo;
- implementare altri criteri di accettazione;
- perfezionare l'algoritmo su istanze piccole;
- minimizzare il numero di valutazioni della fitness;

References

[1] An Effective Algorithm for Minimum Weighted Vertex Cover Problem

[2] Two approximation algorithm for Vertex Cover

[3] A fast heuristic for the minimum weight vertex cover problem

[4] An Effective Algorithm for Minimum Weighted Vertex Cover Problem

[5] A memory-based iterated local search algorithm for the multi-depot open vehicle routing problem

[6] A fast euristic for the minimum weight vertex cover problem