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 13, 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.
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.

Sono presenti all'interno della repository 4 versioni diferrenti del main.

2 Primo approccio

Le informazioni delle istanze dal file `.txt` sono state estratte per mezzo della classe `CustomGraph`. È stata importata ed utilizzata la libreria `NetworkX` per visualizzare graficamente le istanze come reti. Una demo del notebook Jupiter è disponibile qui

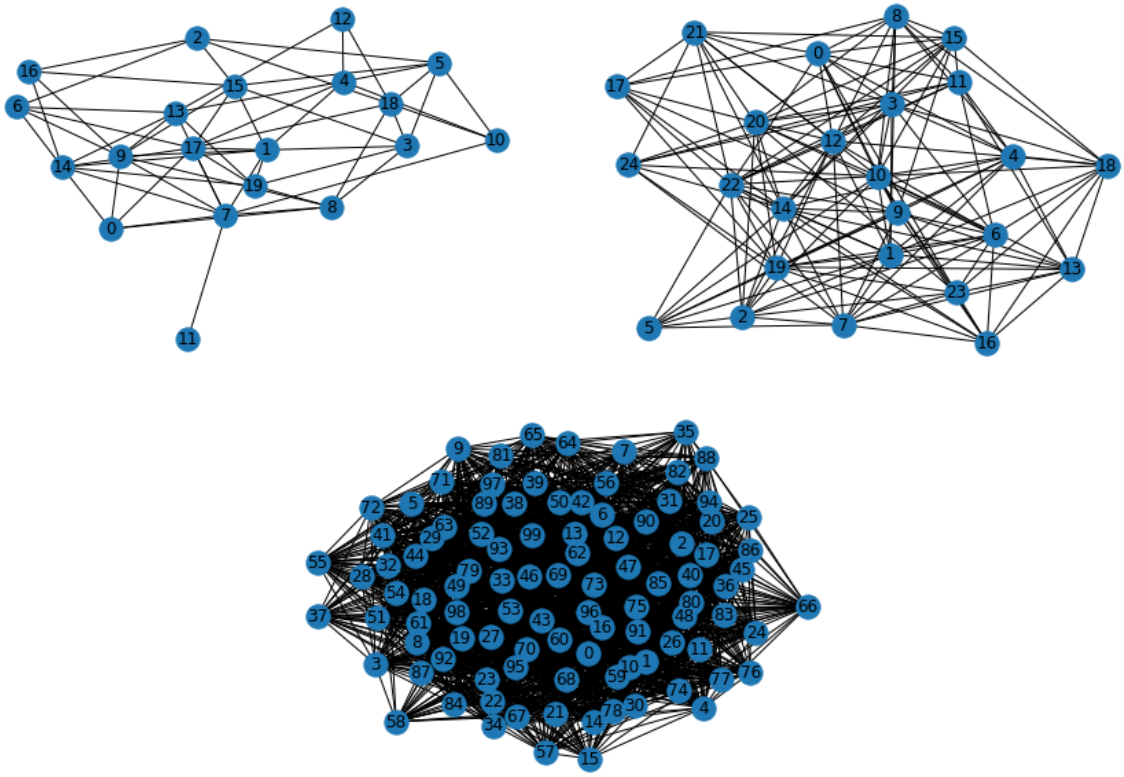

Figure 1: Visualizzazione grafica delle istanze fornite

Utilizzando l'apposita funzione di conversione `cnv_txt_instance`, saranno estratti dall'istanza rispettivamente:

- i nodi con il relativo peso e vicinato;
- numero totale di nodi presenti in un'istanza.

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);

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.

In particolare si nota che la prima versione dell'algoritmo performa bene su SPI ma tende a non funzionare al crescere del numero di nodi, si è tentato dunque di trovare una funzione performasse bene su tutte le istanze: Le diverse versioni sono state testate con delle istanze ridotte rispetto a quelle fornite, in modo da velocizzarne i test. Nel dettaglio:

- **versione A**: costruisce una soluzione iniziale a partire da un nodo scelto randomicamente, l'operatore di perturbazione seleziona randomicamente dalla lista di 'inesplorati' un nodo aggiungere alla soluzione e dai nodi selezionati ne rimuove uno randomico;
- **versione B**: come A, costruisce una soluzione iniziale partendo da un nodo randomico dell'istanza e l'operatore di perturbazione seleziona un numero randomico di nodi da aggiungere e rimuovere dalla soluzione;
- **versione C**: a differenza di A e B, parte dalla soluzione peggiore, contenente tutti i nodi dell'istanza e l'operatore di perturbazione si occuperà di rimuovere un numero randomico di nodi selezionati, fino al massimo la metà dei nodi selezionati;
- **versione D**: come C, parte dalla soluzione peggiore, contenente tutti i nodi dell'istanza e l'operatore di perturbazione si occupera di rimuovere ed aggiungere un numero di nodi randomico: per cercare di ottimizzare la risoluzione di problemi LPI rimuove al massimo un numero che va da 1 alla metà dei nodi selezionati ed aggiunge al massimo 1 o 0 due nodi randomici alla soluzione

Tutti i test sono stati effettuati con criterio `SelectBestCandidateRandomly` spiegato nel dettaglio nei paragrafi successivi. I risultati migliori sono stati ottenuti dalla versione `D` (cartella benchmark test). Di seguito, esempio dei grafici di convergenza, sull'istanza `vc_20_60_05.txt`:

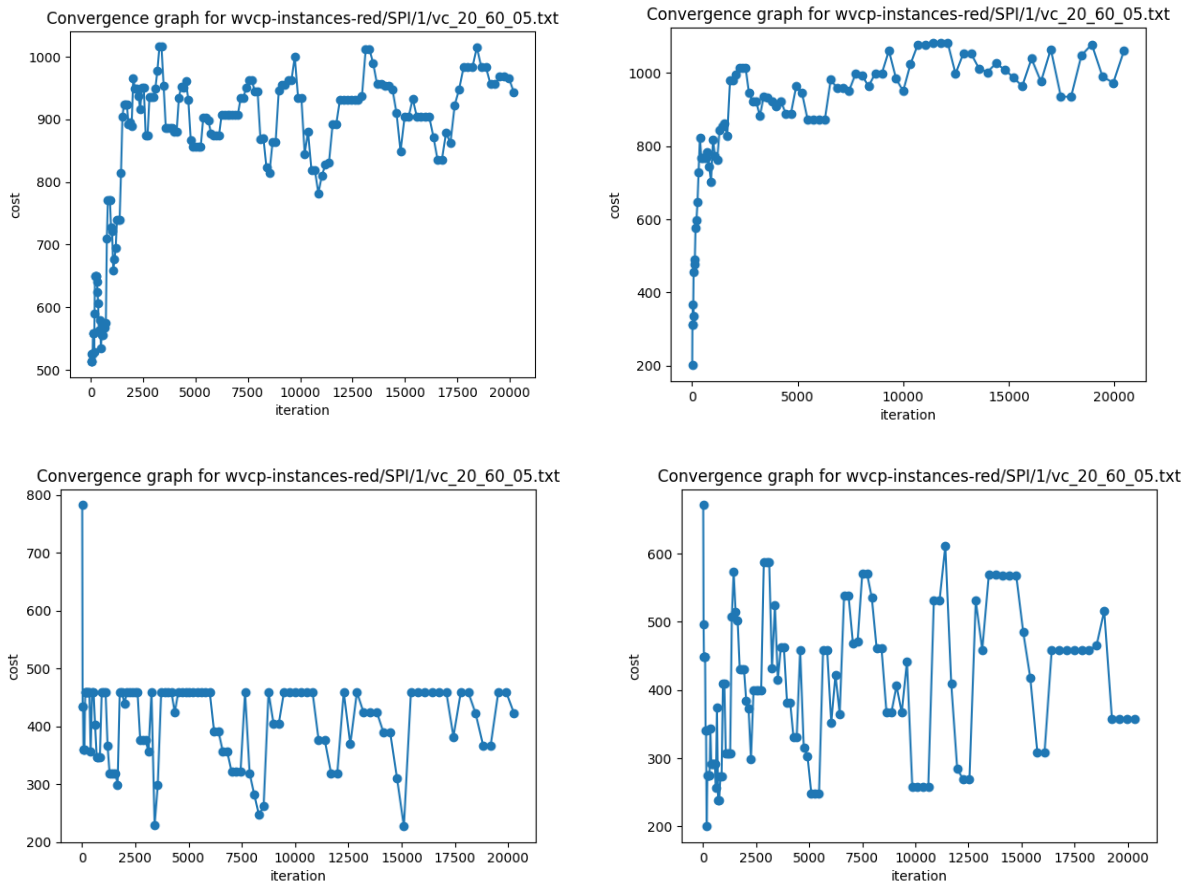

Figure 2: Sample grafici di convergenza

Gli pseudocodici dei due ILS sono i seguente:

Algorithm 1 Iterated Local Search - A, B

```
Require: list of all nodes of instances
  InitialSolution = create initial solution from random node
  CurrentSolution = Apply LocalSearch to InitialSolution
  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
```

Algorithm 2 Iterated Local Search - C, D

```
Require: list of all nodes of instances
  CurrentSolution = Solution with all available nodes selected
  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, di conseguenza si è optato per rimuovere un singolo nodo alla soluzione data.

Sono stati proposti delle varianti usati tra le diverse versioni dell'ILS.

Gli pseudocodici sono i seguenti:

Algorithm 3 Perturbation - A

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

Algorithm 4 Perturbation - B, C

```
Require: Solution
  random perturbation = get random number from (1, length of selected nodes / 2)
  for i in range(0, random perturbation) do
      remove random node from already selected nodes
      add random node from unselected list
  end for
      return perturbed solution
```

Algorithm 5 Perturbation - D

```
Require: Solution
  pert nds remv = get random number from (1, length of selected nodes / 2)
  pert nds add = get random number from (1, 2)
  for i in range(0, perturbed removed nodes) do
      remove random node from already selected nodes
  end for
  if  perturbed solution has unreached nodes then
     for i in range(0, perturbed added nodes) do
         add random node from unselected nodes list
     end for
  end ifreturn perturbed solution
```

In particolare la versione `D` è stata pensata per le istanze LPI. 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

Una delle difficoltà trovate durante l'implementazione è stata la determinazione del criterio di selezione del nodo nella *LocalSearch* di cui è stato trovato opportuno implementarne una versione generica, in modo tale da avere la possibilità di cambiare il criterio facilmente.

Algorithm 6 LocalSearch

```
Require: Solution, BestCandidateCriterio
  while solution is complete do
     best candidate = BestCandidateCriterio()
     add best candidate to solution
  end while
  return solution
```

Volendo ottimizzare il tempo di esecuzione, dopo numerosi test, è stato trovato opportuno procedere con una ricerca First Improvement: durante la prima fase è stata implementata la Random Selection che funziona piuttosto bene su istanze piccole, ma al crescere del numero di nodi da esplorare peggiora. Mentre non è stato considerato vantaggioso effettuare una esplorazione su tutti i nodi prima di effettuare la scelta (best improvement).

Generalmente, un nodo può essere più propenso ad essere aggiunto alla soluzione in base al:

- peso;
- numero di nodi vicini: 'pagare' il costo di un nodo più costoso ma con un elevato numero vi vicini, potrebbe essere più efficiente che selezionare nodi con costo inferiore da cui si otterrà lo stesso vicinato

Sono state implementate delle metriche per valutare il miglior nodo da inserire all'interno della soluzione data una lista di nodi candidati:

Algorithm 7 SelectBestCandidateRandomly

```
Require: list of nodes
  sort list of nodes (randomly) by lower weight or major number of neighbors
  return first element of ordered list
```

Algorithm 8 SelectBestCandidateWeight

```
Require: list of nodes
  sort list of nodes by lower weight
  return first element of ordered list
```

Algorithm 9 `SelectBestCandidateNeighborhood`

```
Require: list of nodes
  sort list of nodes by higher number of nodes
  return first element of ordered list
```

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.

Algorithm 10 `Acceptance criteria`

```
Require: current solution, perturbed solution, term memory
  prob1 = get random number from 0 to 5
  if  prob1 mod 2 == 0 then
     if  term-memory has at least 2 element then
        prob2 = get random number from 0 to 1
        if  prob2 == 0 then return  select one randomly in term memory
        else return  solution with less cost in term-memory
        end if
     end if
  else
     if cost of perturbed solution < cost of current solution then return perturbed
  solution
     end if
  end if
  return random choice between (perturbed solution, current solution)
```

7 Benchmarks e conclusione

È stato implementato uno script che permette di salvare i benchmark le soluzioni ottenute dalle istanze su file `.xls`. La tabella riassuntiva dei benchmarks delle quattro varianti di ILS è consultabile al seguente link). È stato inoltre implementato un metodo per visualizzare i grafici di convergenza dell'algoritmo sulle istanze utilizzando `matplotlib`. (ne saranno importati solo alcuni per alleggerire la repository)

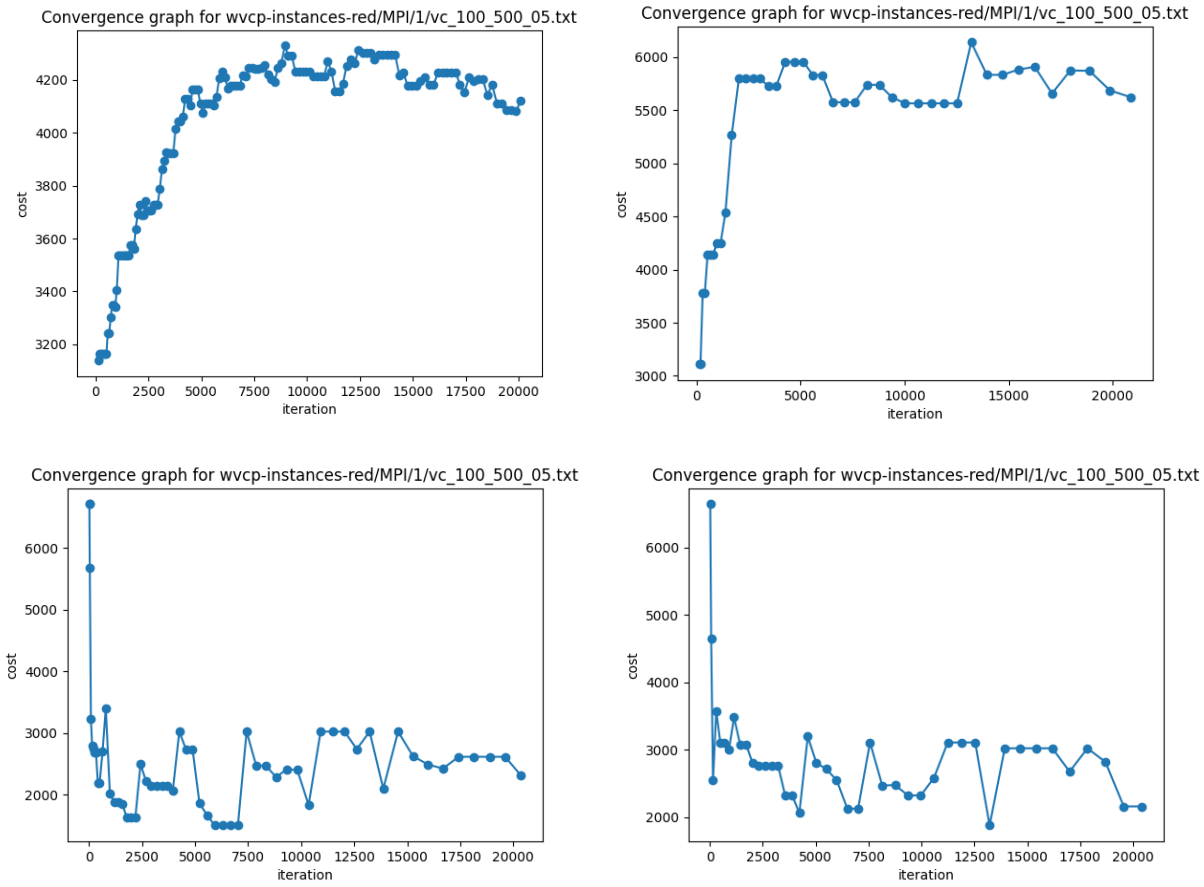

Figure 3: Sample grafici di convergenza delle quattro varianti di ILS su istanza

Guardando i grafici di convergenza e , si deduce che le ultime due versioni presentate performano meglio rispetto alle ultime due: in particolare `ILS versione D` sembrerebbe performare meglio rispetto ad `ILS versione C` nonostante in alcuni casi i valori della soluzione finale trovati siano peggiori. **Si nota inoltre che eliminare dal grafico di convergenza le prime su alcune soluzioni trovate su istanze SPI ed MPI ne renderebbe più agevole la visualizzazione.**

Quest'ultimo è stato eseguito, su tutte le istanze con i tre criteri di selezione del candidato elencati precedenemente. I risultati migliori sono stati ottenuti con `SelectBestCandidateWeight`.

Instance txt	Worst solution	Best solution cost	Mean evaluations for each sol
vc_20_60_01.txt	1286	366	302
vc_20_60_02.txt	1585	352	317
vc_20_60_03.txt	1313	262	350
vc_20_60_04.txt	1263	162	314
vc_20_60_05.txt	1352	459	359
vc_20_60_06.txt	1426	263	319
vc_20_60_07.txt	1573	400	364
vc_20_60_08.txt	1526	320	346
vc_20_60_09.txt	1595	502	362
vc_20_60_10.txt	1416	425	368
MEANS	******	351,1	340,1
vc_20_120_01.txt	1253	311	278
vc_20_120_02.txt	1416	279	256
vc_20_120_03.txt	1319	113	287
vc_20_120_04.txt	1414	71	259
vc_20_120_05.txt	1293	228	236
vc_20_120_06.txt	1271	147	256
vc_20_120_07.txt	1317	160	251
vc_20_120_08.txt	1520	95	250
vc_20_120_09.txt	1587	122	266
vc_20_120_10.txt	1452	117	272
MEANS	******	262,147619	302,4809524
vc_25_150_01.txt	1785	398	266
vc_25_150_02.txt	1561	251	264
vc_25_150_03.txt	1767	205	320
vc_25_150_04.txt	1923	343	288
vc_25_150_05.txt	1792	302	272
vc_25_150_06.txt	1651	370	329
vc_25_150_07.txt	1977	379	293
vc_25_150_08.txt	1693	243	321
vc_25_150_09.txt	1815	285	322
vc_25_150_10.txt	1547	291	297
MEANS	******	276,0702381	300,8306548

Instance txt	Worst solution	Best solution cost	Mean evaluations for each sol
vc_100_500_01.txt	6739	1583	685
vc_100_500_02.txt	7205	2192	805
vc_100_500_03.txt	6764	1870	836
vc_100_500_04.txt	6865	1716	771
vc_100_500_05.txt	7185	3428	812
vc_100_500_06.txt	7228	1358	614
vc_100_500_07.txt	7117	1566	815
vc_100_500_08.txt	6788	1164	755
vc_100_500_09.txt	6894	2053	656
vc_100_500_10.txt	6670	1925	597
MEANS	******	1885,5	734,6
vc_100_2000_01.txt	7027	457	355
vc_100_2000_02.txt	6853	391	379
vc_100_2000_03.txt	6680	315	403
vc_100_2000_04.txt	6922	368	359
vc_100_2000_05.txt	7236	360	348
vc_100_2000_06.txt	6876	500	379
vc_100_2000_07.txt	7322	494	375
vc_100_2000_08.txt	7163	398	425
vc_100_2000_09.txt	6829	312	378
vc_100_2000_10.txt	7238	426	374
MEANS	******	1179,119048	564,552381
vc_200_750_01.txt	14160	4674	1023
vc_200_750_02.txt	13504	3988	1034
vc_200_750_03.txt	14509	4061	1112
vc_200_750_04.txt	13230	3633	1008
vc_200_750_05.txt	14615	6433	1253
vc_200_750_06.txt	13673	5643	1280
vc_200_750_07.txt	13197	4263	1274
vc_200_750_08.txt	13676	4844	1271
vc_200_750_09.txt	13830	4817	1047
vc_200_750_10.txt	13914	5800	1065
MEANS	******	2315,519345	743,3485119
vc_200_3000_01.txt	13915	1196	594
vc_200_3000_02.txt	13790	1125	785
vc_200_3000_03.txt	14245	1129	667
vc_200_3000_04.txt	15202	1172	735
vc_200_3000_05.txt	14095	749	609
vc_200_3000_06.txt	13731	1162	784
vc_200_3000_07.txt	14247	828	682
vc_200_3000_08.txt	14197	1238	761
vc_200_3000_09.txt	14400	1169	689
vc_200_3000_10.txt	13907	857	571
MEANS	******	2024,119498	730,4069975

Instance txt	Worst solution	Best solution cost	Mean evaluations for each sol
vc_800_10000_01.txt	56442	4970	1562
vc_800_10000_02.txt	56442	5425	1487
vc_800_10000_03.txt	56442	5844	1688
vc_800_10000_04.txt	56442	4845	1853
vc_800_10000_05.txt	56442	6063	1706
vc_800_10000_06.txt	56442	5385	1421
vc_800_10000_07.txt	56442	5194	1320
vc_800_10000_08.txt	56442	5752	1563
vc_800_10000_09.txt	56442	6758	1539
vc_800_10000_10.txt	56442	5590	1585
MEANS	******	5582,6	1572,4

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

- implementare altri criteri per la selezione del nodo;
- perfezionare l'algoritmo su istanze medie-grandi;
- testare operatori di perturbazione più robusti;
- incrementare le istanze fornite, in particolare la classe LPI.

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