NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Design of Algorithms for Optimization Problems

2020/2021

Report Phase 2:

Minimum Dominating Set

Teacher:

Pedro Barahona

Authors:

Alexandru Caramida nº 45604

Gonçalo Mateus nº 51927

Min. Dominating Set

Given an undirected graph G=(V, E) we want to compute a minimum size dominating set, a subset S ⊆ V of its vertices such that for all vertices v ∈ V, either v ∈ S or a neighbor u of v is in S.

Computing a dominating set of minimal size is NP-hard.

Model and Seed

The seed is random, select a random vertex and set it to 1.

```
int n = mat.getSize(0);
range Vertices = 1..n;
range D = 0..1;

int t1 = System.getCPUTime();
Solver<LS> ls();
   ConstraintSystem<LS> S(ls);
   /**
      Array of all vertices with values 0 or 1
      0 means it doesn't belong to the dominating set
      1 means it belongs
   */
   var{int} s[i in Vertices] (ls, D) := 0;

   //Seed, start with a random vertex set to 1
   select(v in Vertices)
         s[v] := 1;

   /**
      For every vertex that is covered it needs to be in the set
      or have a neighbour that is, this means for all vertices of s
      the sum of the vertex with the sum of all its neighbors has to be 1 or bigger.
   */
   forall (i in Vertices) {
      S.post(1 <= s[i] + sum(j in Vertices:  mat[i, j] > 0) (s[j]) );
   }

   // Function for the size of the set
   FunctionExpr<LS> F( sum (v in Vertices) s[v] );

   /**
      Otimisation function
      gives more importance to covering all vertices than minimizing the set (ratio 11:10)
   */
   Function<LS> G = 11*S + 10*F;

   // Pointer for the violations
   var{int} violations = S.violations();

ls.close();
```

Meta-Heuristics

Tabu Search

```
int maxit = 2*n;
int it = 0;
int change = 1;
int max_change = n/2;
int best = violations; int best_s = n; int best_it = it;
int tbl = 4; int tblMin = 2; int tblMax = 10; int tabu[Vertices] = 0;
Solution solution(ls);
int nonImprovingSteps = 0; int maxNonImproving = 100; int restartFreq = 1500;
while( (it < maxit && change <= max_change) && ( best_s != optimum || best > 0) ) {
   int old = violations;
   /**
      choose the vertex that leads to the smallest number of violations (the minimal
      delta), if we change s[vertex] to 0 when it was 1 and vice versa.
      only non-tabu assignments are considered.
   */
   selectMin(i in Vertices, delta = G.getAssignDelta(s[i], 1-s[i]): tabu[i] <= it || delta + violations < best) (delta){
      //update tabu and s[i]
      tabu[i] = it + tbl;
      s[i] := 1-s[i];

      // make tabu length change according to violations
      if (violations < old && tbl > tblMin)
         tbl--;
      if (violations >= old && tbl < tblMax)
         tbl++;

      cout << " change var " << i << " to "<< s[i] <<"; delta = "<< delta << endl;
      cout << "-- iteration = " << it << "; violations = " << violations <<"; set size = "<< F.evaluation() << endl;

   }
   /**
      If the new assignment improves the best solution found, then the new assignment is stored into
      the Solution object solution, variable best is updated to reflect the new best number of violations,
      and the number of non-improving iterations is reset to 0
   */
   if (violations < best || (violations == 0 && F.evaluation() < best_s)){
      best = violations;
      best_s = F.evaluation();
      best_it = it;
      solution = new Solution(ls);
      nonImprovingSteps = 0;
      if(violations == 0)
         cout << "best so far of " << best << " with size " << best_s <<" at iteration " << best_it << endl;endl;
   }
   /**
       Check if the maximum limit for non-improving iterations has been reached,
       currently best solution, stored in variable solution is restored and the
       number of non-improving iterations is reset to 0
   */
   else if (nonImprovingSteps == maxNonImproving) {
      solution.restore();
      nonImprovingSteps = 0;
   }
   /**
      Counter for non-improving is increased
   */
   else {
      nonImprovingSteps++;
      change++;
   }

   /**
      Check if restartFreq is reached and no solution found
      if true restart with a new seed
   */
   if (it !=0 && (it % restartFreq == 0) && (best > 0) ) {
      // change a randomm vertex
      with delay(ls)
         forall (i in Vertices)
            s[i] := 0;
      select(v in Vertices)
         s[v] := 1-s[v];
      best = S.violations();
      best_s = F.evaluation();
      solution = new Solution(ls);
   }

   it++;
}
```

Variable Neighbourhood Search

```
int d = 1;
int sol[Vertices];

int maxit = 2*n;
int max_change = n/2;
int it = 0;
int best = violations; int best_s = n; int best_it = it;

while( (it < maxit && d <= max_change) && ( best_s != optimum || best > 0) ) {
// diversify - select a new neighbourhood
   forall(i in 1..d)
      select(v in Vertices){
         s[v] := 1-s[v];
      }
// greedy search for a local optimum
   bool improvement = true;
   while (improvement) {
      selectMin(i in Vertices, delta = G.getAssignDelta(s[i], 1-s[i])) (delta){
         if (delta < 0){
            s[i] := 1-s[i];
         } else
            improvement = false;
         }
      it = it + 1;
   }
// keep best and prepare diversification
   if (violations < best || (violations == 0 && F.evaluation() < best_s)){
      d = 1;
      best = violations;
      best_s = F.evaluation();
      best_it = it;
      forall(i in Vertices)
         sol[i] = s[i];
      cout << "best so far of " << best << " with size " << best_s <<" at iteration " << best_it << endl;
   } else
      d = d + 1;

   it++;
}
int t2 = System.getCPUTime();
```

Test Instances

Social Network Samples

This is a set of anonymised social network samples from

https://davidchalupa.github.io/research/data/social.html

pokec_500.col
pokec_2000.col
pokec_10000.col
pokec_20000.col
pokec_50000.col
gplus_500.col
gplus_2000.col
gplus_10000.col
gplus_20000.col
gplus_50000.col

This is a set of Graph Coloring Instances from

https://mat.gsia.cmu.edu/COLOR/instances.html

anna.col
homer.col
david.col
huck.col

Results

Results sampled from 5 runs per test and all times are in seconds.
We had to make changes to the file_io to import Dimacs and the optimal size.
The heuristics tests stop if iteration limit of 2 * vertices is reached or a limit of changes to the best value found is reached or if it finds the optimal dominating set.
The tests with 20000 and 50000 were not tested, because of their size it was not possible to import the graph to comet.
We weren't able to produce results with the Linear Programming algorithm for the graphs with 50000 vertices because the compute times were too big and we decided to stop the computation.

Run time

Graph	Greedy	Linear P.A.	Tabu	VNS
Samples Pokec	avg	avg	avg	avg
pokec 500	0.0060	0.0289	0.0376	0.0348
pokec 2000	0.0055	0.0736	0.2686	0.2278
pokec 10000	0.0529	44.5557	6.7376	4.0064
pokec 20000	0.2021	623.6569	-	-
pokec 50000	1.6045	-	-	-
Samples Google+				
gplus 500	0.0003	0.0108	0.0502	0.0344
gplus 2000	0.0042	0.0459	0.8748	1.9158
gplus 10000	0.0937	1.7602	28.1312	66.3374
gplus 20000	0.3707	146.3740	-	-
gplus 50000	2.7593	-	-	-
DIMACS Graphs				
anna	0.0001	0.0024	0.0062	0.0096
homer	0.0011	0.0124	0.0722	0.0535
david	0.0001	0.0015	0.0096	0.0032
huck	0.0001	0.0016	0.0064	0.0062

Dominating Set

Graph	Greedy			Linear P.A.	Tabu			VNS		
Samples Pokec	min	max	avg		min	max	avg	min	max	avg
pokec 500, γ=16	16	16	16	16	16	16	16	16	16	16
pokec 2000, γ=75	75	75	75	75	75	75	75	75	75	75
pokec 10000, γ=413	413	414	413.60	413	413	413	413	413	413	413
pokec 20000, γ=921	925	928	926.20	921	-	-	-	-	-	-
pokec 50000, γ≥2706	2768	2781	2774	-	-	-	-	-	-	-
Samples Google+	min	max	avg		min	max	avg	min	max	avg
gplus 500, γ=42	42	42	42	42	42	42	42	42	42	42
gplus 2000, γ=170	175	179	176.80	172	171	171	171	171	172	171.6
gplus 10000, γ=861	894	902	896.40	870	862	864	862.60	866	867	866.6
gplus 20000, γ≥1716	1806	1813	1809.40	1718	-	-	-	-	-	-
gplus 50000, γ≥4566	4835	4847	4840.40	-	-	-	-	-	-	-
DIMACS Graphs	min	max	avg		min	max	avg	min	max	avg
anna, γ=12	12	12	12	12	12	12	12	12	12	12
homer, γ=96	96	96	96	96	96	96	96	96	96	96
david, γ=2	2	2	2	2	2	2	2	2	2	2
huck, γ=9	9	9	9	9	9	9	9	9	9	9

Analysis

Various types of tests were used, both big, small and varying number of edges, for a better comparison between the algorithms and the heuristics. We let the heuristics run after it finds a solution in order to improve it, so in most cases the first solution found is not optimal and most of the compute time is spent improving it.
Looking at all algorithms and heuristics in terms of compute time it's clear that the Greedy is by far the fastest one, we can also observe that the Linear Programming algorithm is the slowest one when it comes to bigger graphs with more edges (pokec has more edges than gplus for the same number of vertices).
The heuristics compute times are not far from the Linear Approximation in the smaller graphs, but in bigger and more complex graphs the Tabu and VNS are faster in graphs of Pokec but slower in gplus. In this case the graphs have the same number of vertices but the Pokec have more edges, a possibility for this behaviour is the increasing compute time at a bigger scale on the Linear Approximation algorithm on graphs with more edges.
In terms of Dominating Set results the algorithms and heuristics are close until more complex graphs are used, at a certain graph size the Greedy algorithm returns worse results compared to the other three. Examples of this observation are the graphs with 10000 vertices or more and gplus 2000. Both the Tabu and VNS heuristics gave better results than the Linear Approximation algorithm on the bigger graphs, with the Tabu having slightly better results for the gplus 10000.
The two heuristics gave the best results and in terms of solution, with the Tabu giving the best ones. So the one to use will depend on the goal and complexity of the problem at hand because none of the four was dominating in both compute time and solution.
In conclusion, the heuristics were easier to work with and give better solutions but are slower than the Linear Approximation algorithm in particular cases, the time could be reduce if we ended the computation when the first solution was found but that would leave a big room for improving the Minimum Dominating Set.