Fiduccia-Mattheyses algorithm. Implementation and modification

Ivan Sladkov, B01-908, MIPT

April 22, 2023

Being an heuristic for solution of a problem of graph partition, Fiduccia-Mattheyses algorithm does not give a completely precise solution, though resulting partition converges to it. Some modifications may be introduced in order to increase convergence speed or precision of partition. In this work original algorithm was implemented and a modification was added to increase speed of algorithm.

1 On algorithm implementation

The original algorithm was implemented with no clustering and with accent on balance. For a formula

$$rW - S_{max} \leq |A| \leq rW + S_{max} \qquad (1)$$

from , which describes soft balancing, coefficient r is taken as 0.5, while maximum divergence between $|A|$ and $|B|$ is $S_{max} = 1$.

Implementation was made in C++. All the containers used in project are based on Standard Template Library. The data structures used are as follows:

- `std::map<int, std::list<int> > gc` – gain bucket
- `std::vector<bool> erased` – shows whether vertex is in bucket or not
- `std::vector<std::pair<int, iterator> > searchSupport` – structure to facilitate and accelerate search

Search through gain buckets is made as $O(1)$.

2 Modification of algorithm

In order to make algorithm converge faster a «cutoff» modification was implemented: if movement of a vertex gives significant cost increase, then further motion is not done. So function `FMPass()` is transformed into:

```
function FMpass
(gain_container, partitionment) :

solution_cost =
partitionment.get cost()

while not all vertices locked {

move = best_feasible_move()

solution_cost-=
gain_container.get gain (move)

if (solution_cost > best_cost+threshold)
then break

gain_container.lock_vertex(
move.vertex())

gain_update (move)

partitionment.apply (move)
}

roll back partitionment
to best seen solution

gain_container.unlock_all()
```

File	Unmodified		Modified	
	Time elapsed	Partition cost	Time elapsed	Partition cost
ISPD98_ibm01.hgr	2311	358	246	1598
ISPD98_ibm02.hgr	2658	520	371	512
ISPD98_ibm03.hgr	2824	3909	429	4148
ISPD98_ibm04.hgr	5345	1882	308	4096
ISPD98_ibm05.hgr	8940	3589	709	6086
ISPD98_ibm06.hgr	4653	1724	751	4799
ISPD98_ibm07.hgr	7191	3018	1299	7274
ISPD98_ibm08.hgr	9761	1382	1360	8776
ISPD98_ibm09.hgr	11944	3873	1995	9556
ISPD98_ibm10.hgr	12128	3008	1637	13001
ISPD98_ibm11.hgr	10075	8693	2313	13463
ISPD98_ibm12.hgr	20845	4622	2959	14155
ISPD98_ibm13.hgr	17317	3222	3225	16767
ISPD98_ibm14.hgr	51533	9833	6173	22643
ISPD98_ibm15.hgr	41726	7899	6496	30221
ISPD98_ibm16.hgr	39586	5539	12863	32235
ISPD98_ibm17.hgr	51904	6354	11020	40035
ISPD98_ibm18.hgr	64940	3483	7323	32570

Table 1: Results of research

3 Comparison

All benchmarks were executed on Apple M2 processor. Result of comparison is shown in table