



An Effective Heuristic for Simple Offset Assignment with Variable Coalescing

Hassan Salamy and J. Ramanujam

Louisiana State University

November 3, 2006

<http://www.ece.lsu.edu/jxr/lcpc06-paper.pdf>



Outline

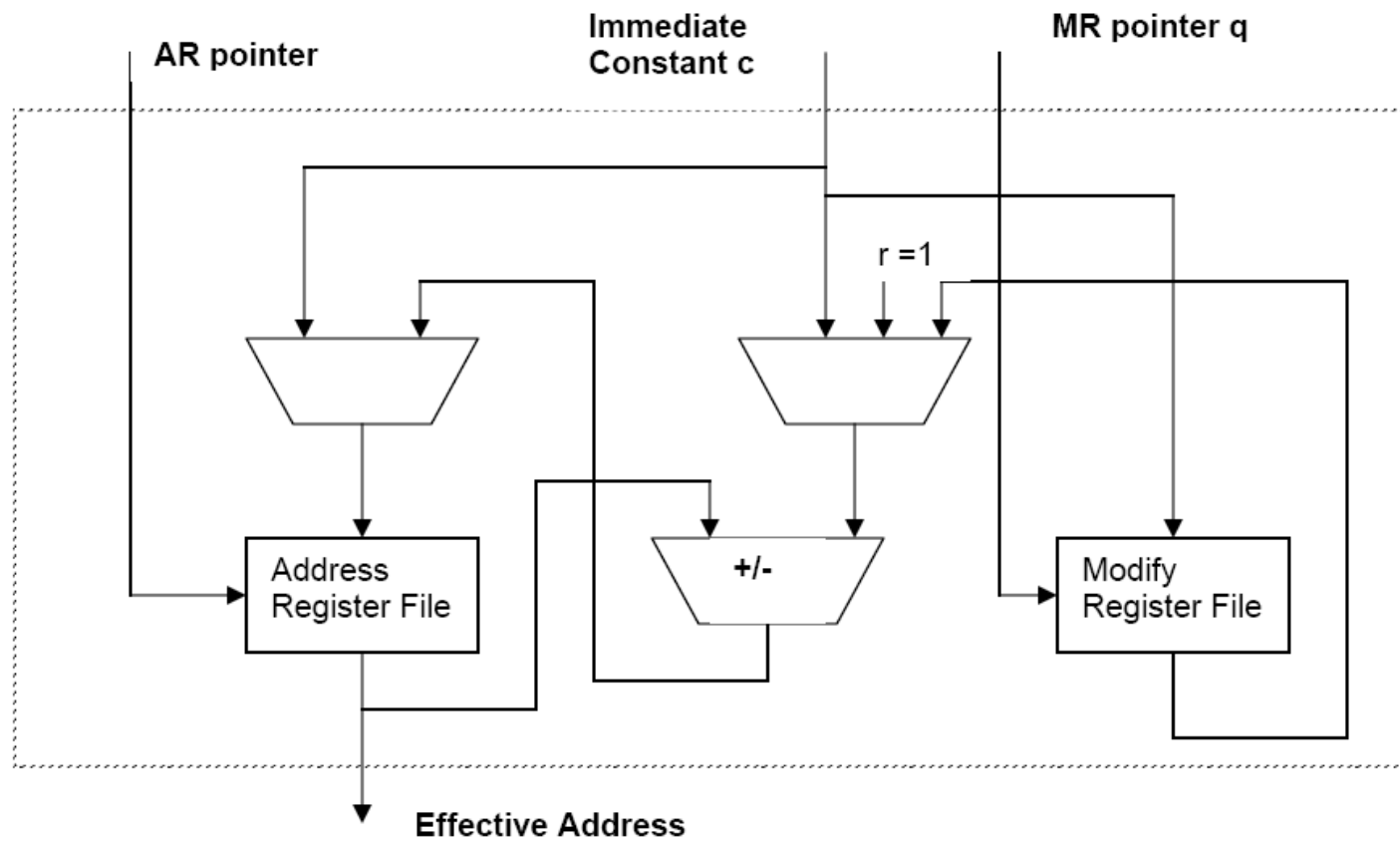
- Introduction
- SOA Example
- SOA with variable coalescing
- Our heuristic
- Example
- Simulated annealing
- Results
- Conclusion



Introduction

- In many Digital Signal Processors with limited memory, programs are loaded in the ROM which directly translates to silicon area.
- Thus the size of the program becomes extremely important.
- Many DSP have address generation unit which consists of mainly address register file, modify register file, and ALU.

Introduction





Introduction

- This architecture supports only indirect memory addressing. So an extra address arithmetic instruction is needed to add an offset to the current address.
- However, this architecture supports auto-increment, auto-decrement without extra instruction.
- Adding/Subtracting one from the current address register can be done in the same load store instruction.



Introduction

- The placement of variables in memory plays a crucial role in the degree of exploiting the auto-increment/decrement.
- Offset assignment is the problem of placing the variables in the memory to maximally utilize auto-increment/decrement.
- Simple offset assignment (SOA) is the offset assignment problem with one address register.
- SOA is NP-complete equivalent to the Maximum Weight Path Cover.



SOA Example

$$a = d + c$$

$$b = e + b + a$$

Access sequence: d c a e b a b

d	c	a	e	b
---	---	---	---	---

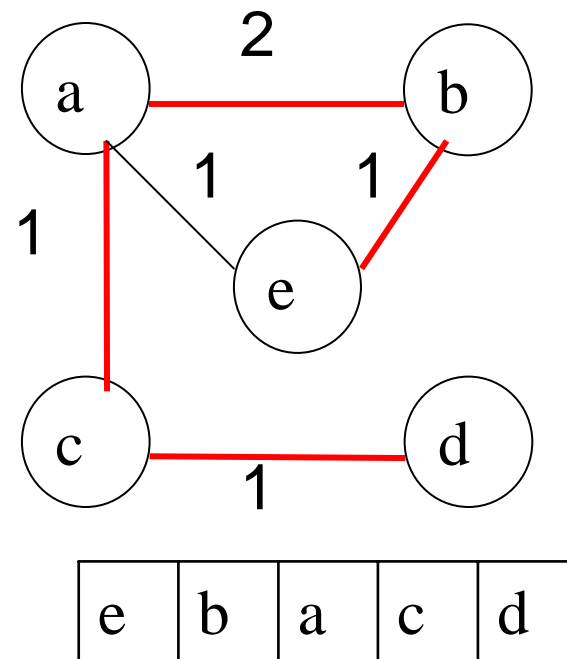
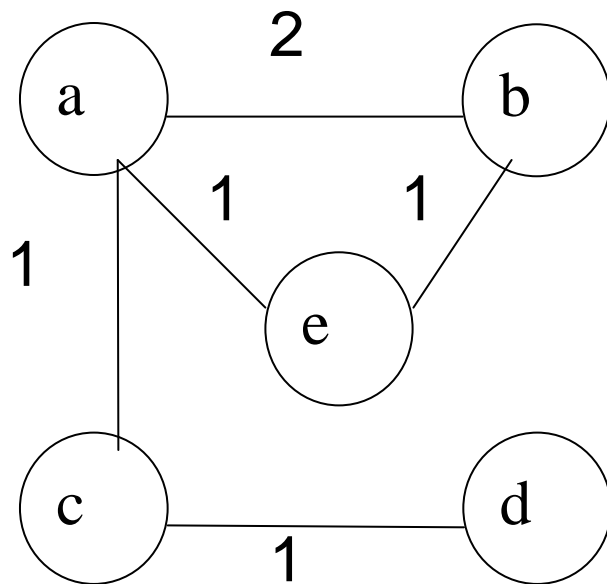
```
LDAR AR0,&d
LOAD *(AR0)+
ADD *(AR0)+
STOR *(AR0)+
LOAD *(AR0)+
ADD *(AR0)
SBAR AR0,2
ADD *(AR0)
ADAR AR0,2
STOR *(AR0)
```

d	c	a	b	e
---	---	---	---	---

```
LDAR AR0, &d
LOAD *(AR0)+
ADD *(AR0)+
STOR *(AR0)
ADAR AR0,2
LOAD *(AR0)-
ADD *(AR0)-
ADD *(AR0)+
STOR *(AR0)
```

SOA Example

The access sequence: d c a e b a b





SOA with variable Coalescing

- In SOA each memory location is assigned only one variable.
- In CSOA, more than one variable can be mapped into the same memory location (coalesced).
- Two variables can be coalesced if their live ranges do not overlap at any time.
- An interference graph (IG) is such that each vertex represents a variable and an edge in IG between two variables means that they interfere and thus they cannot be coalesced.



SOA with Variable Coalescing

- Two variables can be coalesced if:
 - they do not interfere;
 - if coalesced, no node in the access graph will end up with degree > 2 considering only already selected edges; and
 - if coalesced, the access graph will still be acyclic considering only selected edges.



Our CSOA Heuristic

- Our algorithm integrates both selection and coalescing option in each step to minimize the number of address arithmetic instructions as well as to minimize the memory requirement for storing variables in memory.
- At each step, it examines the coalescing candidates and picks the one with the maximum Gain.
- Also it picks the edge with the maximum weight as a candidate for selection at this step.
- Only edges (a, b) such that (a, b) belongs IG will be considered candidates for selection.



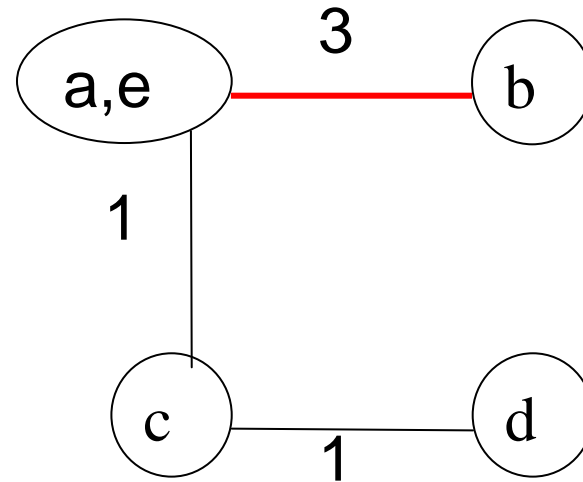
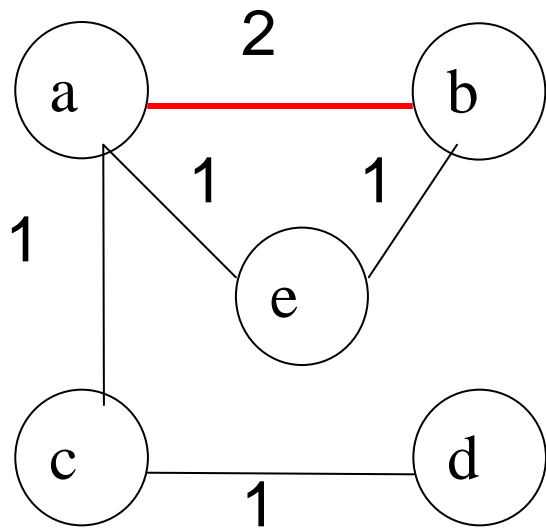
Our CSOA Heuristic

$$Gain(a,b) = \frac{Actual_Gain(a,b)}{Possible_Loss(a,b)}$$

$$Actual_Gain(a,b) = W(a,b) + \sum_{\substack{x \in Adj(a) \cap Adj(b) \\ (b,x) \in Selected_edges \\ (a,x) \notin Selected_edges}} W(a,x) + \sum_{\substack{y \in Adj(a) \cap Adj(b) \\ (b,y) \notin Selected_edges \\ (a,y) \in Selected_edges}} W(b,y)$$

$$Possible_Loss(a,b) = \sum_{\substack{(a,x) \notin IG, (b,x) \in IG \\ (b,x) \notin Selected_edges}} (a,x) + \sum_{\substack{(b,y) \notin IG, (a,y) \in IG \\ (a,y) \notin Selected_edges}} (b,y) + 1$$

Our CSOA Heuristic



$$\text{Actual_Gain}(a,e) = 2$$



Our CSOA Heuristic

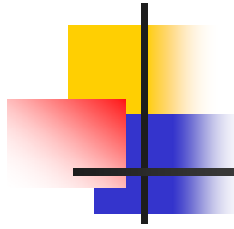
Tie-break functions:

$$T1(a, b) = \text{degree}(a) + \text{degree}(b)$$

$$T2(a, b) = \sum_{x \in \text{Adj}(a)} W(a, x) + \sum_{y \in \text{Adj}(b)} W(b, y)$$

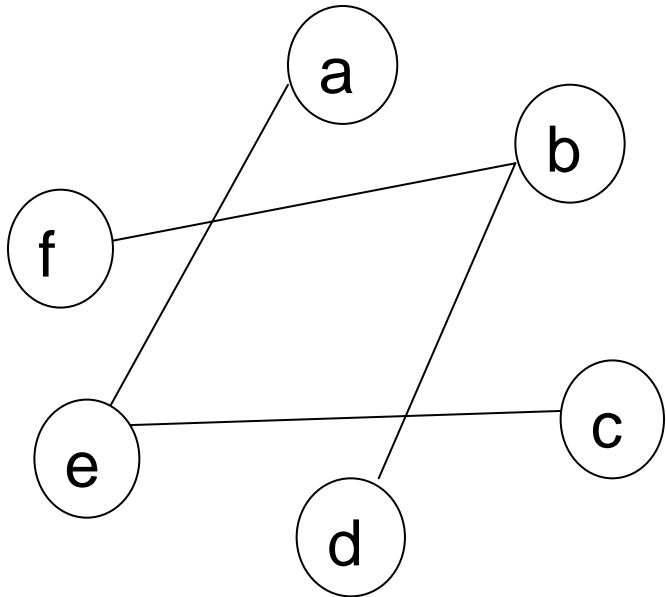
Selection: T1 then T2 if needed

Coalescing: Actual_Gain, T1, then T2

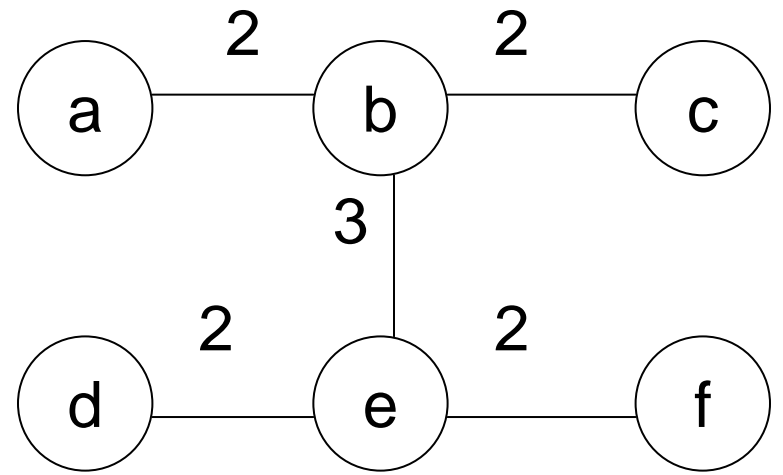


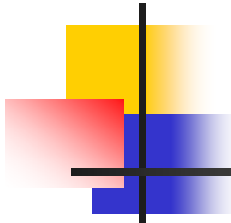
Example

Interference Graph:



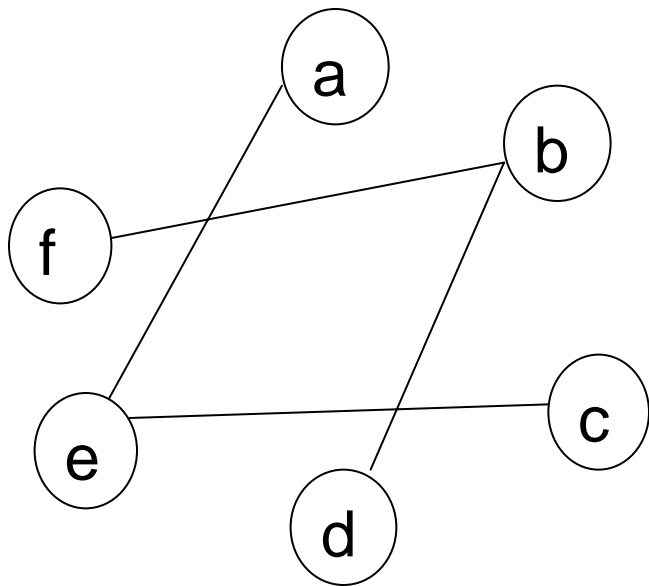
Access Graph:



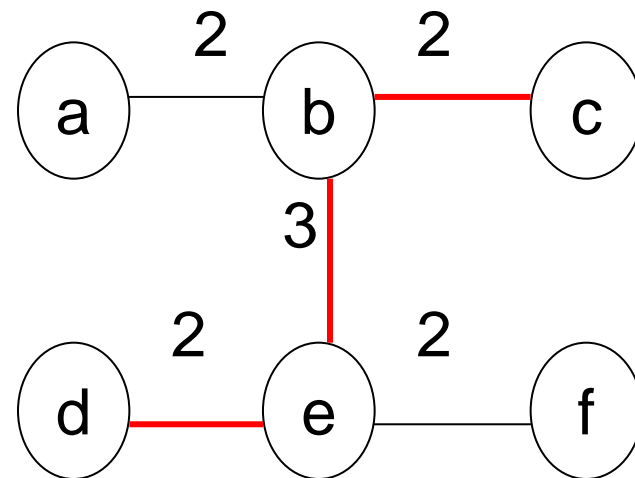


Example

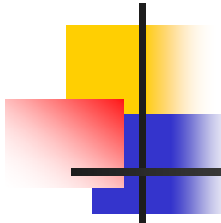
Interference Graph:



Liao's solution:

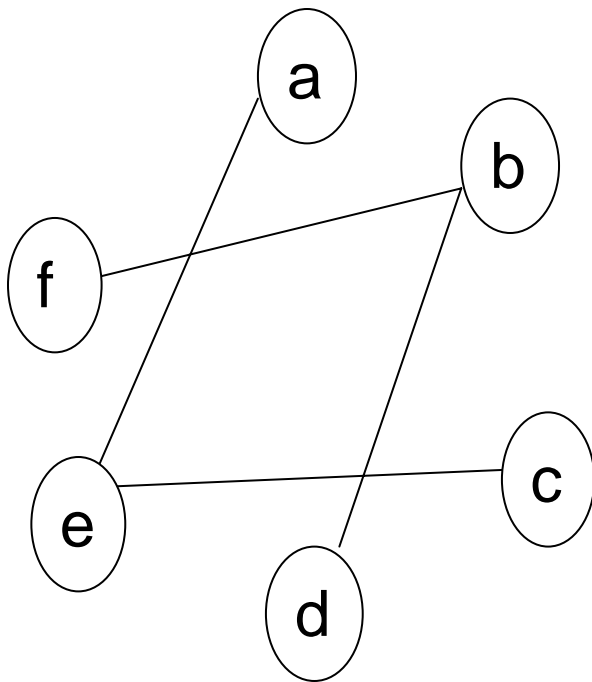


Cost = 4

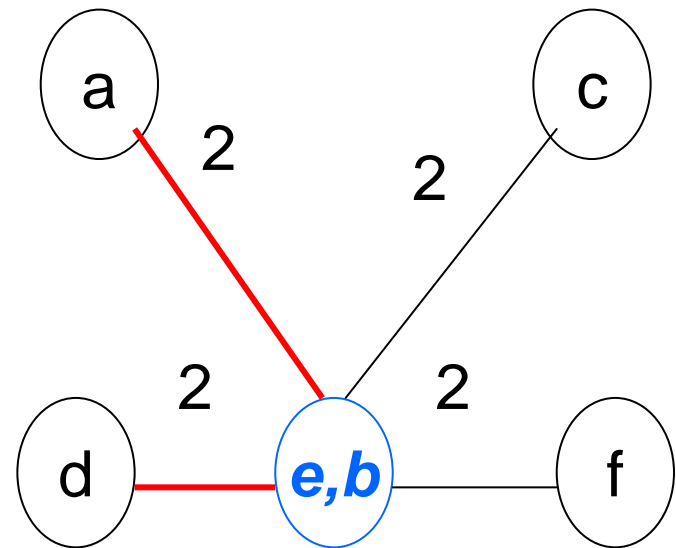


Example

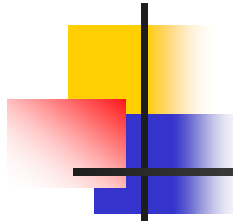
Interference Graph:



Example of a bad coalescing candidate (e,b):

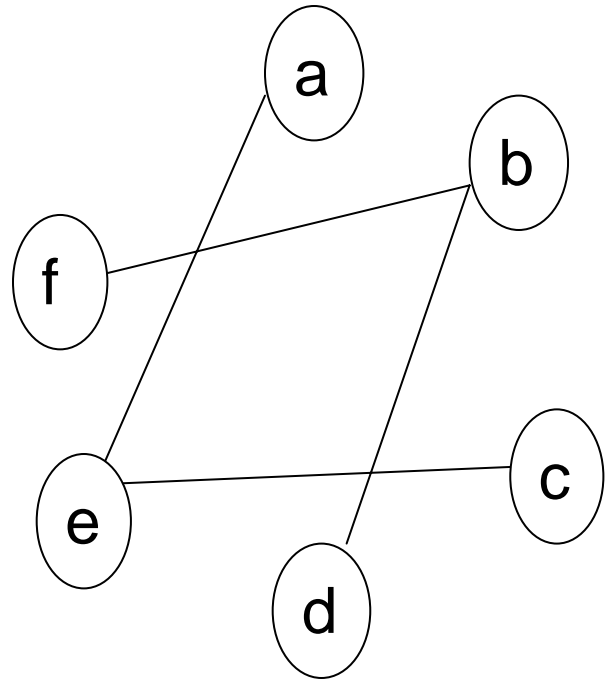


Cost = 4

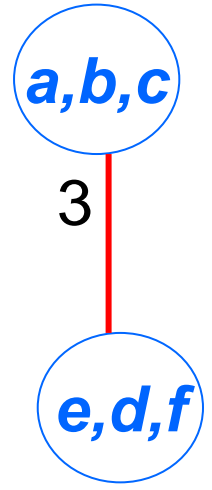


Example

Interference Graph:



Optimal Solution by our heuristic:

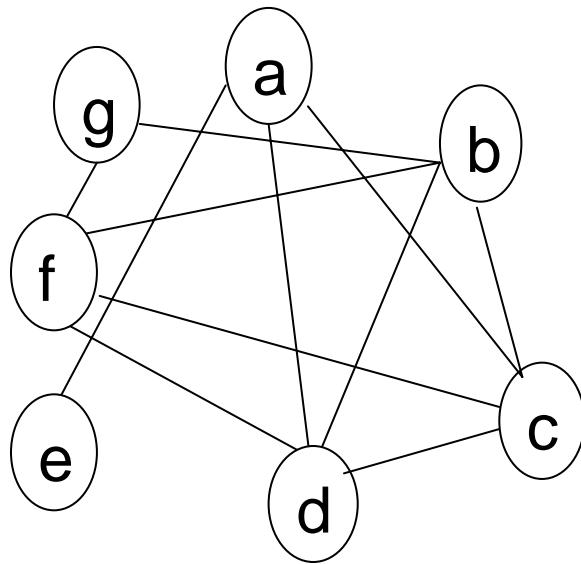


Cost = 0

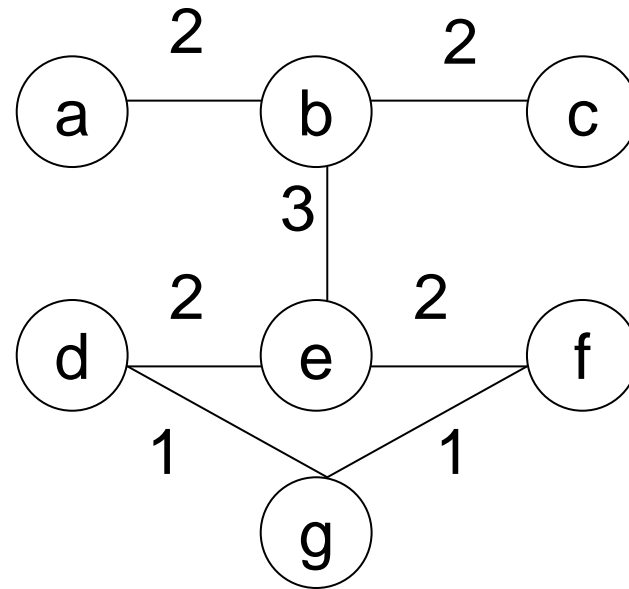
1. **Coalescence_SOA_Algorithm {**
2. **Input: Access sequence; Interference graph IG.**
3. **Output: Offset assignment.**
4. **Build the access graph (AG) from the access sequence**
5. **L = list of edges (x,y) such that (x,y) belongs IG in decreasing order of their weights.**
6. **Coalesce = false**
7. **Select = false**
8. **Do{**
9. **Find a pair of nodes (a,b) for coalescing with maximum Gain that satisfies the validity conditions.**
10. **If such a pair of nodes is found, then Coalesce = true.**
11. **Among the edges that belong to L pick the first edge (c,d) that satisfies the validity conditions.**
12. **If such an edge is found, then Select = true; remove (c,d) from L.**
13. **If (Coalesce &&Select)**
14. **If(Actual_Gain(a, b) ≥ Weight(c, d))**
15. **Update access graph AG.**
16. **Update interference graph IG**
17. **Update list L**
18. **Else**
19. **Select edge (c,d)**
20. **Else if (Coalesce)**
21. **Update access graph AG**
22. **Update interference graph IG**
23. **Update list L**
24. **Else if(Select)**
25. **Select edge (c,d)**
26. **} While (Coalesce || Select)**
27. **Return offset assignment**
28. **}**

Example

Interference Graph:

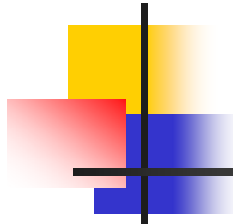


Access Graph:



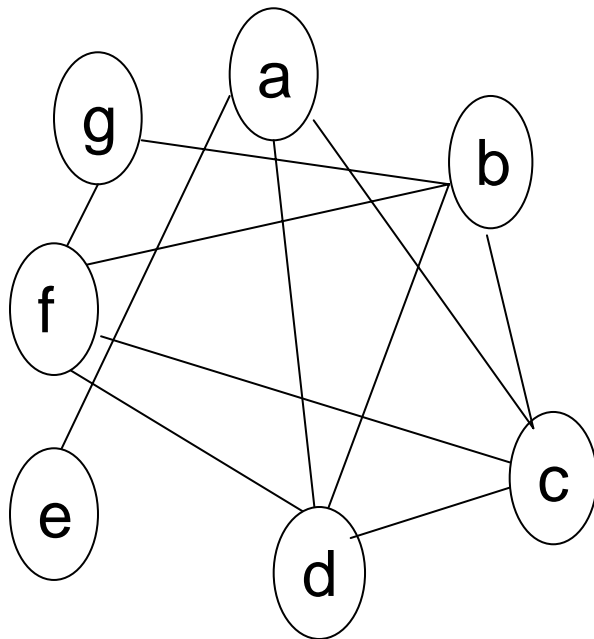
Best Coalesce candidate: $\text{Gain}(a,b) = 2/2$

Best Select candidate: $w(b,c) = 2$

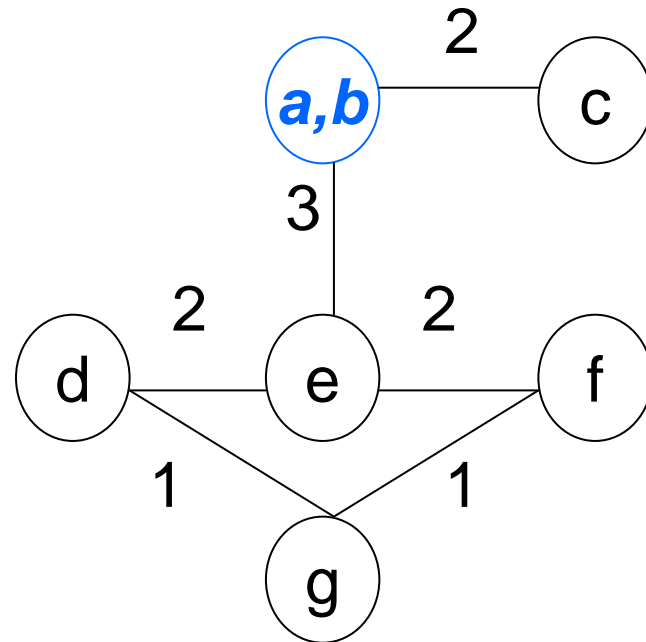


Example

Interference Graph:

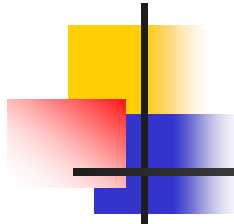


Access Graph with a,b coalesced:



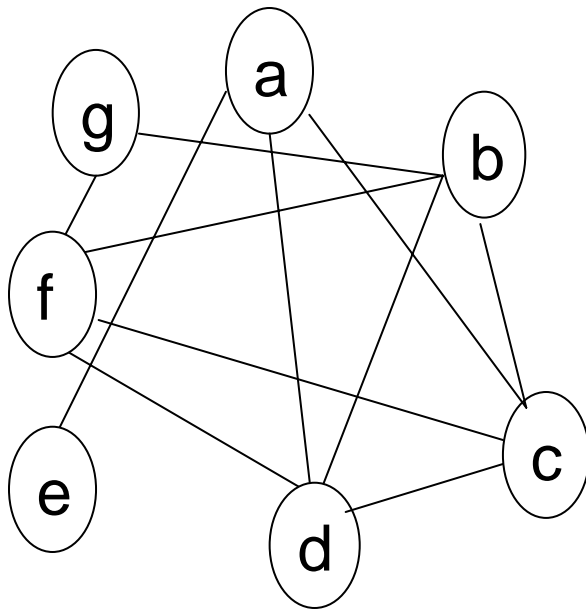
Best Coalesce candidate : $\text{Gain}(d,e) = 2/2$

Best Select candidate : $w(ab,e)=3$

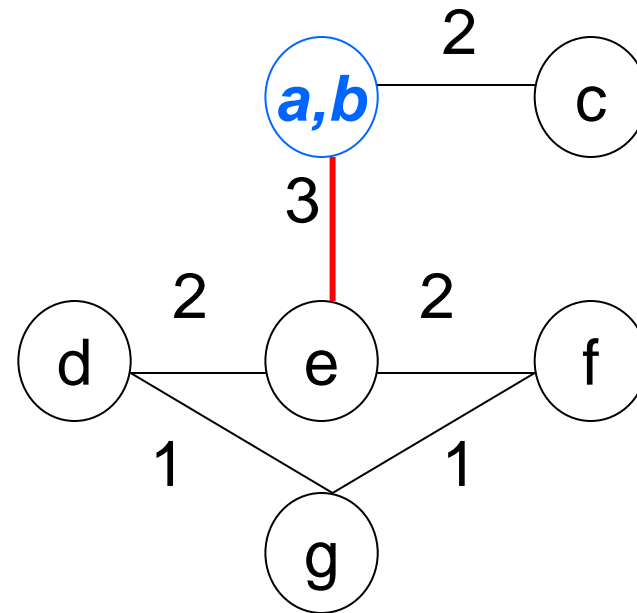


Example

Interference Graph:

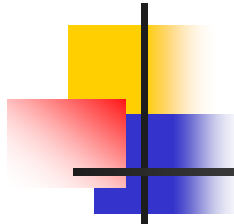


Access Graph with (ab,e) selected:



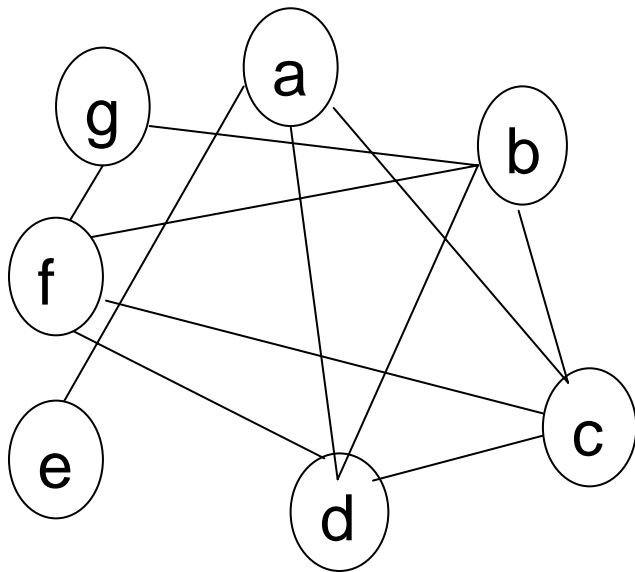
Best Coalesce candidate: $\text{Gain}(d,e) = 2/2$

Best Select candidate: $w(ab,c)=2$

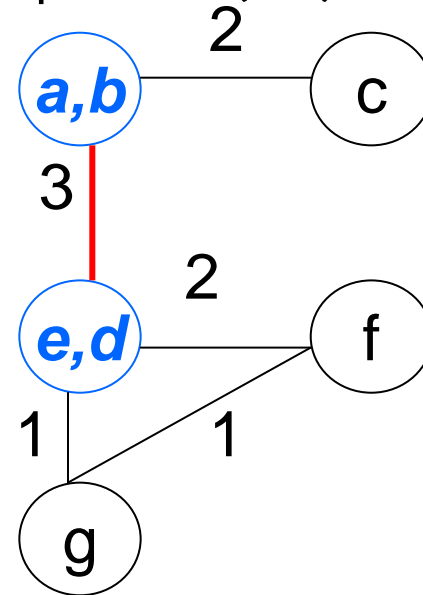


Example

Interference Graph:

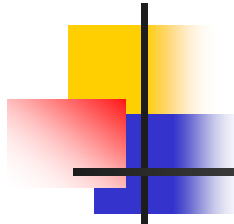


Access Graph with (e,d) coalesced:



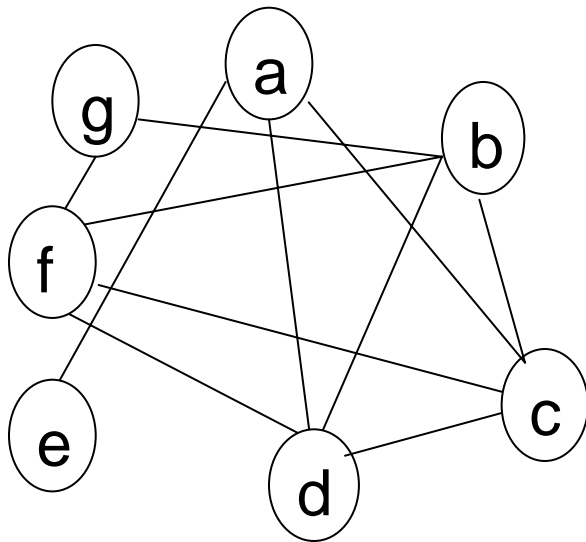
Best Coalesce candidate: $\text{Gain}(ed,g) = 1$

Best Select candidate: $w(ab,c)=2$

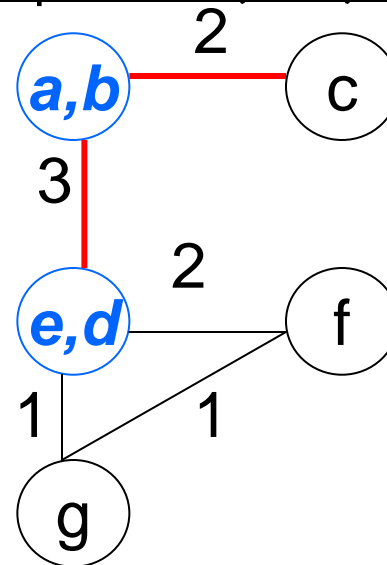


Example

Interference Graph:

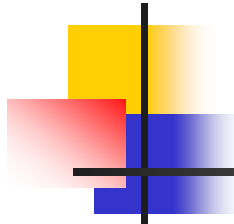


Access Graph with (ab,c) selected:



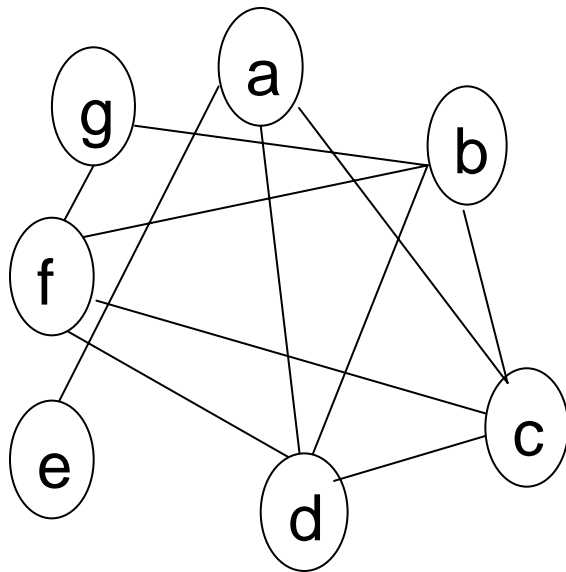
Best Coalesce candidate: $\text{Gain}(ed,g) = 1$

Best Select candidate: $w(ed,f) = 2$

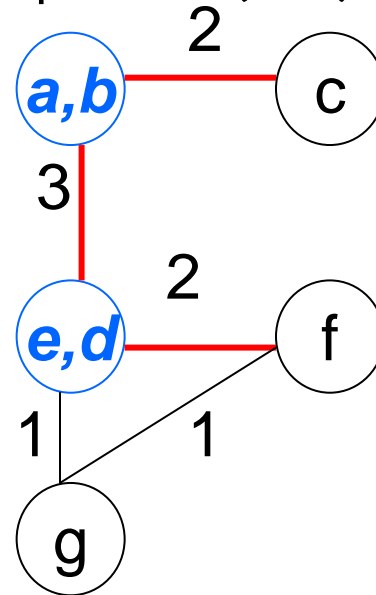


Example

Interference Graph:

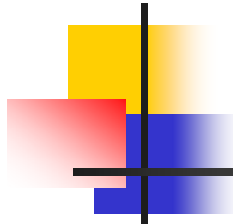


Access Graph with (ed,f) selected:



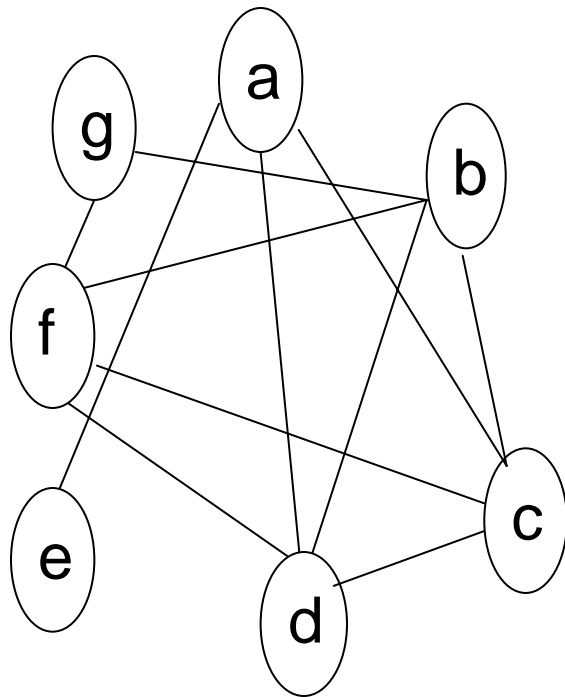
Best Coalesce candidate: $\text{Gain}(ed,g) = 2/1$

Best Select candidate: $w(g,f)=1$

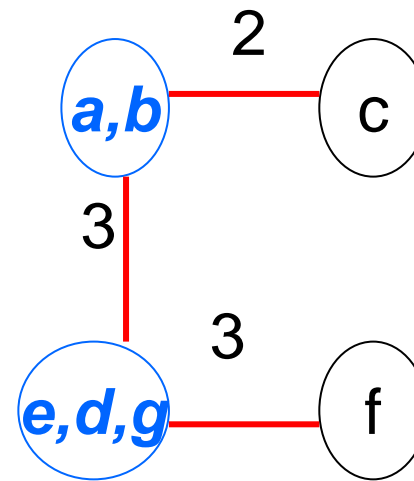


Example

Interference Graph:



Final Solution: Offset Assignment:



c
<i>a,b</i>
<i>e,d,g</i>
f



Simulated Annealing

- Simulated Annealing (SA) is a global stochastic method that is used to generate approximate solutions to very large combinatorial problems.
- The annealing algorithm begins with an initial feasible configuration, and then a neighboring solution is created by perturbing the current solution.
- If the cost of the neighboring solution is less than that of the current solution, the neighboring solution is accepted; else it is accepted or rejected with a certain probability.
- The probability of accepting inferior solutions is a function of a parameter, the temperature T , and the change in cost between the neighboring solution and the current solution.



Simulated Annealing

Neighbor function that we used will perform one of the following:

- 1- Exchange the content of two memory locations.
- 2- Move the content of one memory location.
- 3- Uncoalesce a coalesced node into two or more nodes.
- 4- Coalesce two memory locations.



Results

Bench marks	TB (%) [LM96]	GA(%) [LD98]	INC-TB(%) [LM96] [Leu03]	CSOA(%) [OOA03]	Our algorithm (%)	SA (%)
adpcm	89.1	89.1	89.1	45.6	42.1	39.1
epic	96.8	96.6	96.6	50.2	47.0	44.9
g721	96.2	96.2	96.2	27.9	26.3	23.2
gsm	96.3	96.3	96.3	19.4	14.8	13.4
jpeg	96.9	96.7	96.7	32.2	31.0	29.1
mpeg2	97.3	97.1	97.2	34.3	31.2	29.9
pegwit	91.1	90.7	90.7	38.8	39.5	36.1
pgp	94.9	94.8	94.8	32.2	29.8	27.4
rasta	98.6	98.5	98.5	21.1	19.9	19.5



Results

Benchmarks	Number of variables	Number of Memory slots in [OOA03]	Number of memory slots using our algorithm
adpcm	198	55	43
epic	4163	1125	767
g721	1152	289	199
gsm	4817	1048	433
jpeg	13690	4778	2555
mpeg2	8828	2815	1503
pegwit	4122	1454	910
pgp	9451	2989	1730
rasta	4040	1056	557



Conclusion

- A heuristic for the SOA with variable coalescing is presented.
- The heuristic was able to decrease the number of explicit address arithmetic instructions and the memory requirement for storing the variables.
- The SA further improved the results
- Future work: General Offset Assignment with Variable Coalescing