Deep Learning-based Hybrid Graph-Coloring Algorithm for Register Allocation (2019)

Das, Dibyendu, Shahid Asghar Ahmad, and Kumar Venkataramanan. arXiv preprint arXiv:1912.03700 (2019). AMD

IIE8557-01 고등강화학습

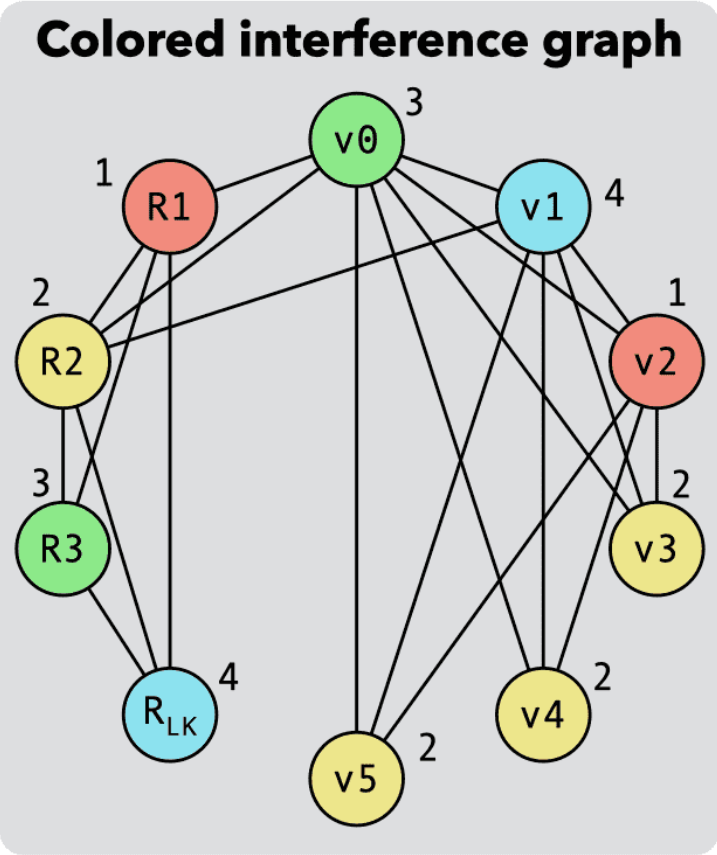
경영과학연구실 이태헌

컴파일러 최적화에서 레지스터 할당이란?

- 컴파일러 최적화 과정 중 레지스터 할당은 프로그램 각 변수를 컴퓨터 레지스터에 할당하는 과정임
- 레지스터는 CPU 내부의 매우 빠른 메모리로, 데이터를 저장하고 연산에 사용됨.
- 프로그램을 실행할 때, 레지스터를 효율적으로 사용하는 것은 성능에 큰 영향을 미침.
- 그러나 레지스터는 제한된 자원이므로 모든 변수를 동시에 레지스터에 할당할 수 없음
- 이때, 어떤 변수들이 동시에 활성화 되어 있지 않다면, 같은 레지스터를 공유할 수 있음.

Colored Interference graph

- Interference graph는 변수들 간 관계를 나타냄
- 각 노드는 변수를 대표하고, 노드 사이의 연결선은 두 변수가 동시에 활성화될 때 존재함



- Graph-coloring algorithm은 각 노드에 색을 할당하는 과정임
- 색상은 레지스터를 상징, 연결된 노드들은 서로 다른 색을 가져야 함
- 즉, 두 변수가 연결선으로 연결되어 있다면, 그 변수들은 동시에 레지스터에 있을 수 없음

가능한 적은 수의 색을 사용하여 모든 노드를 채색하는 것이 목표
(최소화의 레지스터를 사용하여 프로그램 실행)

‘레지스터 할당 과정에서 효율적인 Colored interference graph Algorithm을 학습하고자 함’

Deep learning based (LSTM) 사용하여 interference graph를 색칠하는 휴리스틱 학습

↓

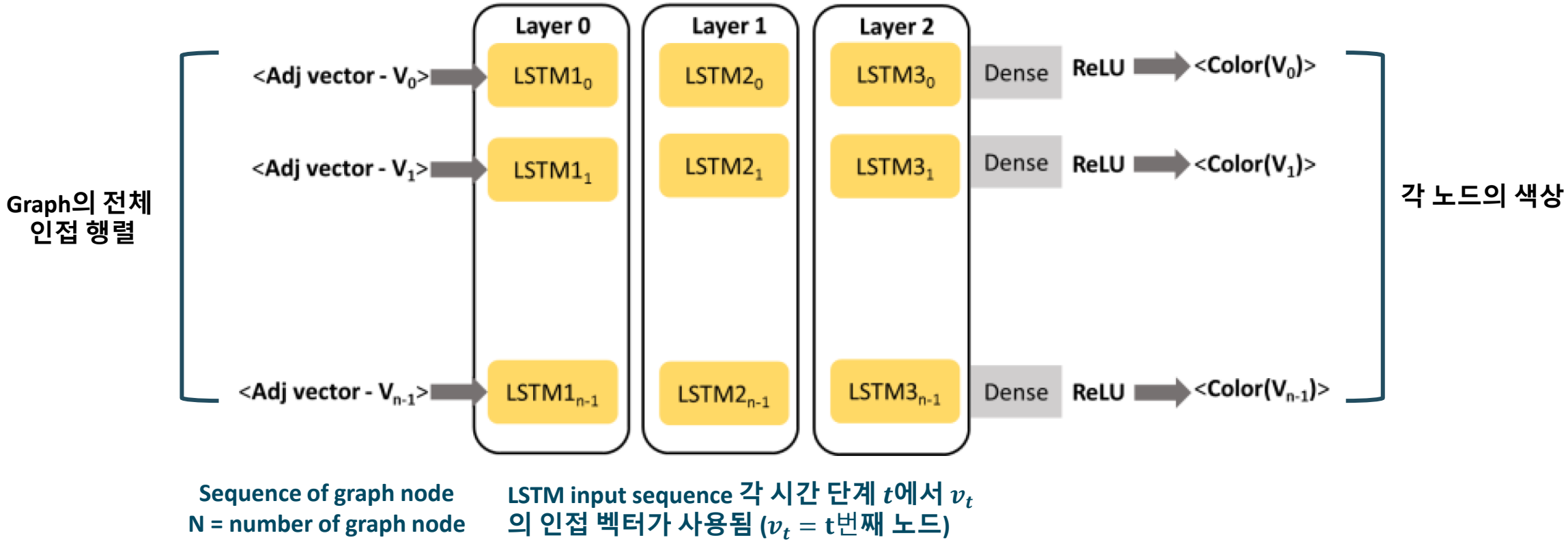
LSTM 만으로는 유효하지 않은 색상 할당을 방지하는 제약 조건 표현이 어려움

↓

색상 할당 후 색상 보정 단계를 추가하여 Hybrid algorithm 구성

LSTM-based Model for Graph Coloring

- 최대 100개 노드로 제한한 그래프를 LSTM 모델을 사용하여 Graph coloring



Training the model

- Very_nauty package 통해 생성된 무작위 graph를 데이터로 사용
- Very_nauty : 무작위 graph 생성과 군집 수 및 색상 수 계산을 목표로 하는 그래프 알고리즘 C 라이브러리

- 이를 위해, graph_gnp, graph_chromatic_number 두 가지 함수를 사용함

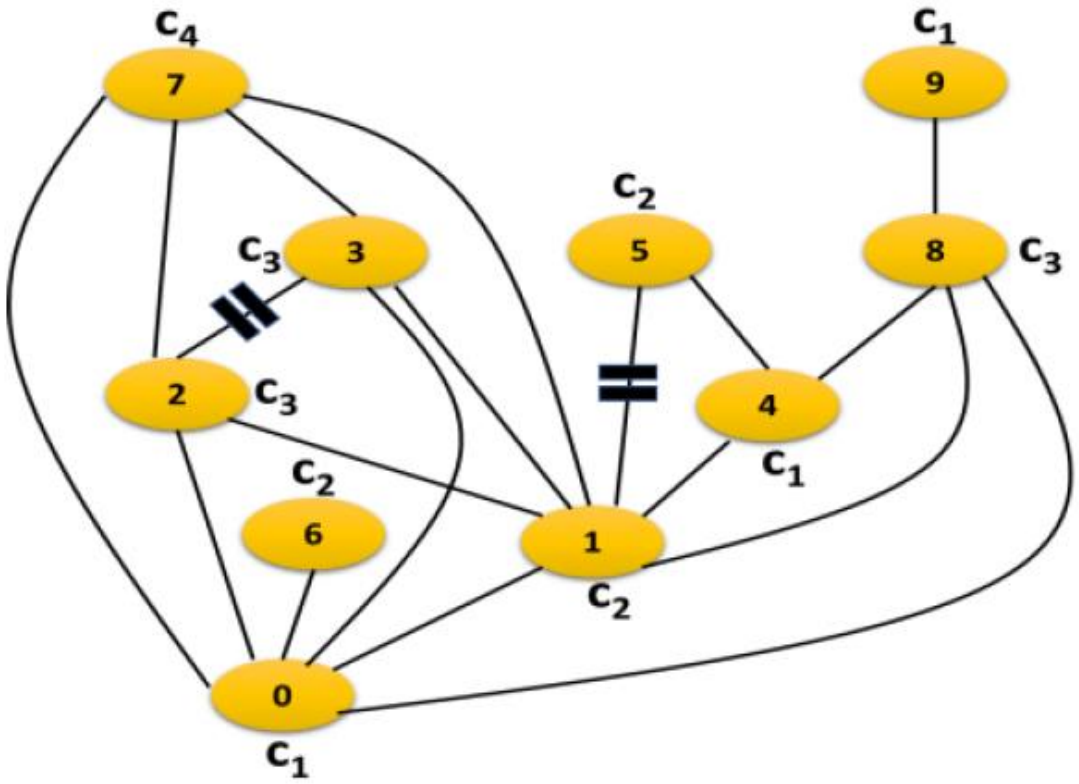
- Graph_gnp : n개 노드의 무작위 graph를 두 노드가 간선으로 연결될 확률(p)로 생성함
- Graph_chromatic_number : graph의 정확한 색상 수 계산, 생성된 무작위 graph의 개별 노드에 대한 색 할당도 추출할 수 있음

.

- 1개의 노드부터 100개의 노드까지 10000개의 가까운 무작위 graph 생성
- 생성된 각 그래프에 대해 Graph_chromatic_number 함수 사용하여 색상 할당

Inference and Color Correction

- **Inference 단계에서는 training 된 모델을 사용하여 새로운 샘플의 각 vertex에 할당된 색상을 예측함**
- **Color correction 단계에서는 새 그래프의 각 노드의 색상을 예측하면, 예측 유효성을 검사함.**
  (각 연결선을 검사하여 두 끝점이 동일한 색상을 가지고 있는지 확인)



- 각 노드는 특정 색 $C_1, C_2, C_3, C_4,$ 로 표시됨
- 노드2, 노드3의 경우 모두 C_3으로 색칠되었으며, 이는 유효한 색칠이 아님
- Color Correction 단계에서는 무효한 연결선을 수정하여 각 끝점이 다른 색을 가지도록 해야함

Inference and Color Correction

- **Color correction 알고리즘**

```
Algorithm: ColorCorrection()
Colors = Set of colors allocated to the graph after inference
for each INVALID edge e=<n1,n2> do {
   Let c=color of the nodes n1 and n2
   for ( c1 in Colors AND c1 != c ) do {
      if ( exists e1 = <n1,M> such that M has color c1 )
         continue;
      Color c1 is not used by any neighbor of n1
      reuse color c1 for n1
   }
   if no color found for reuse for n1 then {
      for ( c2 in Colors AND c2 != c ) do {
         if exists e1 = <n2,M> such that M has color c2
            continue;
         Color c2 is not used by any neighbor of n1
         reuse color c2 for n1
       }
   }
   if no color found for reuse for both n1 AND n2 then {
      Create new color cn
      Colors = Colors U {cn}
       Assign cn to n1
   }
}
```

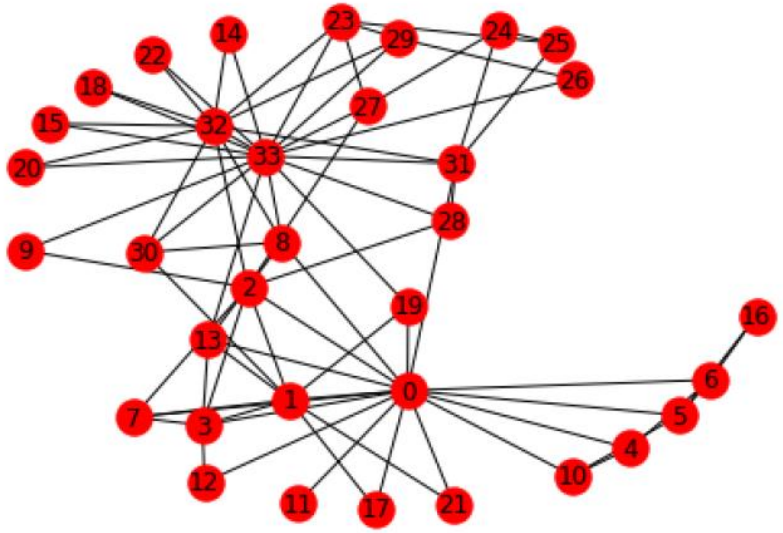
무효한 연결선 $e =< n1, n2 >$ 각각에 대해 아래 단계를 수행함

1. $n1, n2$ 의 색상 c 를 확인
2. c와 다른 색상 c 1을 찾아 $n1$ 이웃 중 c 1 을 사용하지 않는지 확인함. 만약 그렇다면, $n1$에 c 1 을 재할당
3. $n1$에 재사용할 색상을 찾지 못하면 $n2$에 대해 같은 과정 수행
4. $n1, n2$ 모두에 대해 재사용할 색상을 찾지 못하여 새 색상 cn 을 생성하고 색상 집합에 추가한 후 $n1$에 할당

Performance on some popular graphs

- **Popular graphs에 제안 모델 적용한 성능 결과**

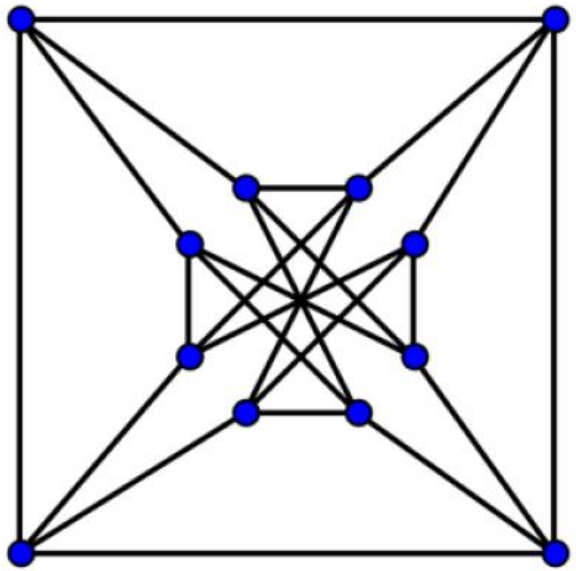
Karate Graph



Inference
4 coloring, 무효한 간선 :23/79

Color correction
5 coloring

Chvatal Graph



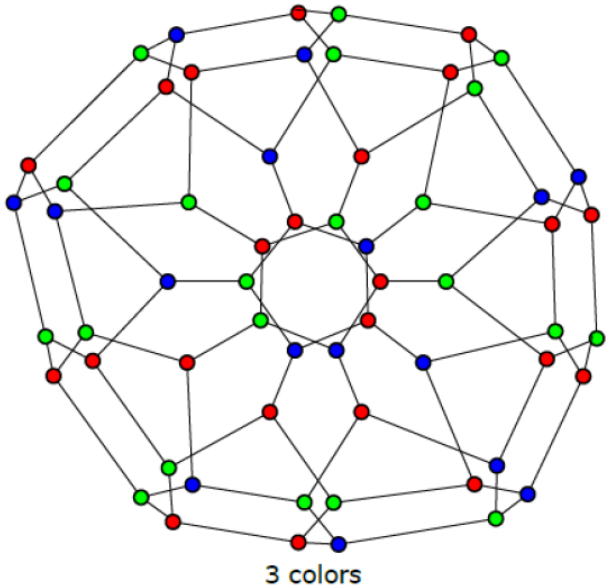
Inference
3 coloring, 무효한 간선 :7/24

Color correction
4 coloring

Performance on some popular graphs

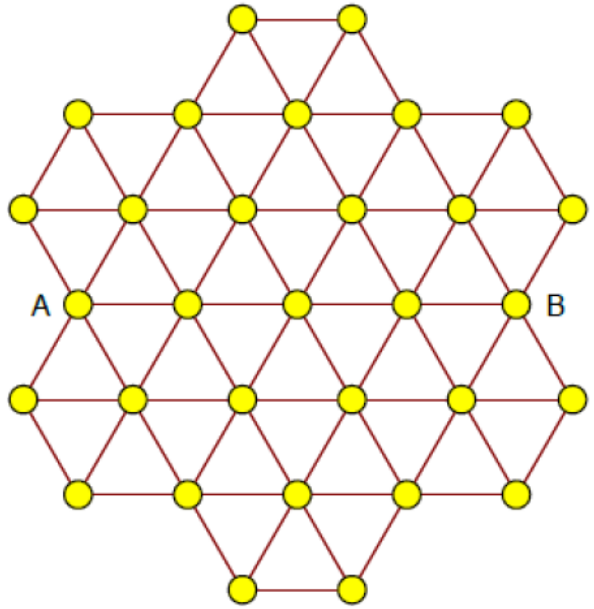
- **Popular graphs에 제안 모델 적용한 성능 결과**

Baidu Graph



Inference
3 coloring, 무효한 간선 :35/90

Color correction
4 coloring



Inference
4 coloring, 무효한 간선 :19/72

Color correction
5 coloring

COLOR dataset results

- **Sparser graphs(insertions2,3, 및 mugg100) 에서는 최적 색상 개수와 일치하는 결과를 보여줌**
- **Dense graphs(queens8_12) 에서는 최적 색상 개수에 비해 5개의 추가 색상을 필요로 함**

- <n,e> : 노드 (n), 연결선(e)
- X(G) : graph 최적 색상 개수

Graph-name	<n,e>	X(G)	Predicted(G) before CC	Predicted(G) after CC	% invalid edges
insertions2	37,72	4	3	4	34%
insertions3	56,110	4	3	4	41%
insertions4	67,232	4	5	6	28%
mugg100	100,166	4	3	4	35%
queens8_12	96,1368	12	9	17	12%
queens5x5	25,160	5	6	9	21%
queens6x6	36,290	7	6	9	16%
queens7x7	49,476	7	7	11	15%
queens8x8	64,728	9	8	12	13%

Results from some SPEC CPU® 2017 benchmarks

- SPEC CPU 2017 benchmarks로부터 function interference graphs를 수집하여 LLVM reg-alloc 와 결과 비교
- LLVM (Low Level Virtual Machine) : 컴파일러의 일부로 실제로 레지스터 할당을 수행하는 프로그램

Functions	LLVM reg-alloc	DL before correction	DL after correction
switch_arcs	17	14	22
replace_weaker_arc	16	10	13
insert_new_arc	14	11	15
resize_prob	7	4	7
marc_arcs	12	10	12
refreshPositions	14	14	14
refreshArcPositions	8	4	7
master	23	13	24
worker	25	16	24
markBaskets	11	9	9
primal_bea_mpp	21	14	26
primal_feasible	9	10	10
flow_org_cost	14	7	10
flow_cost	13	8	10
refresh_neigbour_lists	10	6	9
update_tree	19	8	19
primal_start_artificial	11	7	9
primal_imnus	7	5	7
write_objective_value	5	5	8
main	6	3	4
TOTAL	262	174	257

- 전체적으로 DL model은 색상 수정 후에 더 많은 레지스터를 사용함
- 하지만, 일부 경우에는 색상 수정 전에 LLVM 보다 더 적은 레지스터를 사용하기도 함
- LLVM 대비 2%의 성능 개선 보임

Results from some SPEC CPU® 2017 benchmarks

- SPEC CPU 2017 benchmarks로부터 function interference graphs를 수집하여 LLVM reg-alloc 와 결과 비교
- LLVM (Low Level Virtual Machine) : 컴파일러의 일부로 실제로 레지스터 할당을 수행하는 프로그램

Functions	LLVM reg-alloc	DL before correction	DL after correction
lzma_index_buffer_decode	11	7	9
index_decode	12	13	15
lzma_index_hash_append	10	10	10
lzma_index_hash_decode	14	16	18
lzma_stream_buffer_decode	14	6	11
stream_decode	14	19	19
lzma_stream_footer_decode	8	6	6
lzma_vli_decode	13	8	13
lz_encoder_prepare	13	6	13
lzma_lz_encoder_init	11	7	9
lz_encode	14	10	15
lzma_mf_hc3_find	21	9	19
...	...	...	...
TOTAL	732	500	715

- LLVM 대비 2.5% 성능 개선을 보임

Conclusions

- 레지스터 할당 문제를 해결하는 문제를 위한 Graph coloring 방법에 딥러닝 프레임워크를 적용하는 방법을 제안함
- 제안 알고리즘은 LSTM 기반 딥러닝 모델을 이용. 후처리 color correction 단계를 포함하는 하이브리드 모델임.
- Popular models, SPEC CPU 2017 benchmark 에서 LLVM에 의해 생성된 interference graph와 비교하여 제안 모델이 최적 할당 또는 상당 기간 조정된 최신 휴리스틱과 비교하여 성능이 뒤쳐지지 않음을 보여줌

Q & A