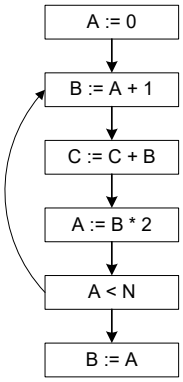
Граф сметњи

Проблем који онемогућава да се исти регистар додели двема привременим променљивама назива се сметња. Најчешћи узрок сметње је преклапање опсега животног века променљивих (али постоје и други разлози: на пример, ако променљива A треба да буде генерисана инструкцијом која не може да адресира регистар R тада постоји сметња између променљиве A и регистра R).

Граф сметњи се често у рачунару представља матрицом сметњи. Колоне и редови матрице су исти и представљају све променљиве у програму. На пресеку одговарајуће колоне и реда биће уписан индикатор да ли постоји сметња између променљиве у колони и променљиве у реду. На слици 1 је дат граф тока управљања једног једноставног програма. Анализом животног века променљивих добијамо резултате који су приказани у табели 1. На основу резултата добијених из анализе животног века променљивих правимо граф сметњи, табела 2.



Слика 1 Граф тока управљања

	USE	DEF	I		II		III	
			OUT	IN	OUT	IN	OUT	IN
6	a	b		a		a		a
5	a		c	ac	ac	ac	ac	ac
4	b	a	ac	bc	ac	bc	ac	bc
3	bc	c	bc	bc	bc	bc	bc	bc
2	a	b	bc	ac	bc	ac	bc	ac
1		a	ac	c	ac	c	ac	c

Табела 1 Излази из анализе животног века променљивих

	A	B	C
A			X
B			X
C	X	X	

Табела 2 Матрица сметњи

Да би се из резултата анализе животног века променљивих направио граф сметњи, користе се два правила:

1. За сваку дефиницију променљиве A, у чвору који није MOVE, додај у граф сметњи нове сметње између променљиве A и сваке променљиве B_i која живи на излазу чвора: $(A, B_1)\ldots(A,B_n)$
2. За MOVE инструкцију A=C, додај графу сметњи нове сметње између променљиве A и сваке променљиве B_i живе на излазу чвора, која није једнака C:$(A, B_1)\ldots(A,B_n)$; где је B_i различито од C.

Додела ресурса

Фазе у компајлеру које претходе фази доделе ресурса подразумевају да на циљаној платформи постоји неограничен број регистара (и ресурса уопште) у које се смештају променљиве. Фаза доделе ресурса има задатак да тај неограничени број регистара сведе на тачно онај број регистара колико има на циљаној платформи: нпр. Mips - 32 регистра. Још један задатак фазе доделе ресурса може бити да након доделе регистара открије MOVE инструкције које пребацују садржај између истих регистара и обрише такве инструкције. Узмимо, на пример, да је пре почетка фазе доделе ресурса у програму постојала инструкција A := B. После фазе доделе ресурса утврђено је да ове променљиве немају никаквих сметњи и променљивој A је

додељен регистар R0, али је и променљивој B додељен регистар R0. Сада уместо A := B у коду стоји инструкција R0 := R0 која је није потребна и треба је избацити.

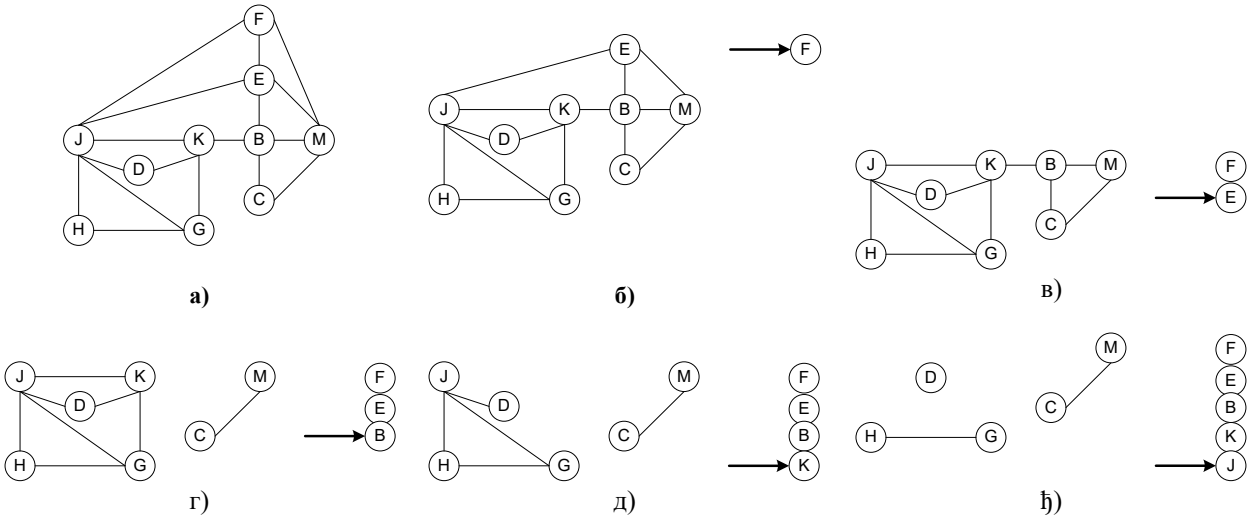
Проблем доделе ресурса се често своди на проблем бојења графа. Код бојења графа је потребно сваком чвору доделити неку боју, али тако да суседни чворови (чворови који су повезани неком ивицом графа) нису обојени истом бојом. У најпростијој форми проблема, циљ је обојити произвољни граф са најмањим бројем боја. Ако нам је граф који желимо да бојимо управо граф сметњи, а боје конкретни ресурси циљне архитектуре, онда је повезаност проблема бојења графа и доделе ресурса јасна. Међутим, код доделе ресурса задатак није обојити са што мањим бројем боја (ресурса), већ је довољно обојити са бројем боја мањим од количине слободних ресурса на циљној платформи (на примеру Мипс архитектуре то би било 32). Овакав проблем нема решење, у смислу да није откривен алгоритам који ће за произвољни граф гарантовати бојење са бројем боја мањим од K (где је K количина расположивих ресурса). Ако одређени алгоритам не успе обојити одређени граф са мање од K боја разлог за то може бити једна од две ствари: или граф ни теоретски није могуће обојити са мање од K боја, или јесте могуће обојити теоретски, али конкретан алгоритам не може то да изведе.

Алгоритам за доделу ресурса који ће у овој вежби бити обрађен састоји се од четири подфазе:

1. Фаза формирања (build): формира се граф сметњи на основу анализе животног века променљивих.

2. Фаза упрошћавања (simplify): граф се боји применом следеће хеуристике: Претпоставимо да чвор M графа G има мање од K суседа. Нека је G` граф који се добија уклањањем чвора M, G` = G – {M}. Очигледно је да ако је могуће обојити G` онда је могуће обојити и G, пошто кад се чвор M дода обојеном графу G`, његови суседи могу бити обојени са највише K-1 боја, тако да увек постоји слободна боја за чвор M. Ово природно води ка рекурзивном алгоритму бојења графа, у ком се чворови чији је ранг мањи од K поступно уклањају из графа сметњи и гурају на стек. Свако упрошћавање доводи до снижавања ранга чвора суседних чвору који се уклања, чиме се ствара могућност за даље упрошћавање. **Питање:** Шта је ранг чвора у графу? Шта је ранг графа?

   На следећој слици дат је пример упрошћавања графа сметњи за процесор који има 4 регистра, дакле K је 4 (а). Редом се бирају чворови који се уклањају из графа. Најпре се бира чвор са највећим рангом мањим од K. На датом примеру графа, чвор F има највећи ранг мањи од 4 (3), зато се ставља на стек и том приликом се брише са графа. Брисање подразумева брисање самог чвора и брисање свих веза тог чвора са његовим суседима (б). Затим се открива да чвор E има ранг 3, исто мањи од 4, тако да се он брише са графа и ставља на стек (в). Остатак примера се лако може испратити. Фаза упрошћавања се завршава када се обришу сви чворови са графа и поставе на стек.



Слика 2 Пример упрошћавања графа сметњи

3. Фаза преливања (spill): у току фазе упрошћавања може се десити да се граф сметњи не може више упростити, а да још увек постоје чворови у њему. То значи да се дати граф не може обојии са K боја. Тада се прибегава растерећивању графа тако што ће неким променљивама бити промењен ресурс, то јест биће пребачене из регистара у, на пример, меморију. Тај поступак се назива преливање, сликовито описујући шта се дешава са преосталим променљивама када је скуп ресурса препуњен – преливају се у меморију. Након овога обично се мора поновити фаза избора инструкција над целим или делом међукода, па затим опет анализа животног века, и на крају поново овај алгоритам доделе ресурса. Оваква петља се може извршити више пута, уколико је граф сметњи превише сложен.

4. Фаза избора(select): након што је граф успешно упрошћен потребно је обојити чворове. Редослед бојења чворова је одређен стеком на који су се чворови постављали приликом упрошћавања графа. Дакле, чворови ће бити бојени обрнутим редоледом од оног којим су скидани са графа. Једино правило које се користи приликом бојења је да се чвору не додели боја која је већ додељена неком од суседних чворова.

Реализација

Приликом реализације задатака потребно је користити постојеће структуре и функције. Типови података описани су у датотеци *Types.h*. Инструкције су описане у структури `Instruction`. У оквиру ове вежбе користи се: тип инструкције (`type`), показиваче на одредишни, први и други операнд (`dst, src1, src2`). Више инструкција описује се помоћу листе у типу `Instructions`.

```
typedef struct InstructionStruct
{
    InstructionType type;

    Variable* dst;
    Variable* src1;
    Variable* src2;
} Instruction;

typedef list<Instruction*> Instructions;
```

Променљиве су описане у структури `Variable` и садрже: назив (`name`), позицију унутар матрице сметњи (`pos`), и боју ресурса (`assigment`). Више променљивих описано је помоћу листе у типу `Variables`.

```
struct Variable
{
    char name;
    int pos;
    Regs assigment;
};

typedef list<Variable*> Variables;
```

Типови инструкција описани су преко набрајајућег типа `InstructionType`, и подржане су следеће инструкције: `T_MOVE`, `T_EQU`, `T_SUB`, `T_ADD`, `T_MUL`, `T_COND`, `T_RET` и `T_OTHER`. Боје ресурса описане су у типу `Regs` и подржани су регистри `reg0`, `reg1` и `reg2`.

Граф сметњи описан је у структури `InterferenceGraph`. Ова структура садржи листу свих променљивих коришћених у инструкцијама (`variables`), матрицу сметњи (`values`) и величину матрице (`size`). Матрица сметњи описана је преко двоструког низа који садржи вредност `__INTERFERENCE__` на местима где постоји сметња између променљивих и `__NO_INTERFERENCE__` на местима где нема сметњи.

```
typedef struct
{
    Variables* variables;
    char** values;
    int size;
} InterferenceGraph;
```

За пример се може узети матрица сметњи из табеле 2. Она садржи 3 променљиве A, B и C. Променљива A има позицију 0, B позицију 1 и C позицију 2. Позиције (**`pos` из структуре `Variable`**) треба искористити за правилно индексирање матрице сметњи. Променљиве у листи `variables`, из структуре `InterferenceGraph`, не морају бити увезане оним редом којим се налазе у матрици сметњи. Нпр. у листи се редом налазе C, A и B, док матрица сметњи (`values`) описује редом сметње између променљивих A, B и C.

Приликом решавања задатка је потребно користити већ постојеће статичке библиотеке: *libResourceAllocation.lib* и *libLivnessAnalysis.lib*.

За рад са инструкцијама потребно је користити функције из датотеке *Instructions.h*. Ту се налазе функције за испис једне или више инструкција у конзолу. Ове функције треба користити приликом откривања грешака у програму, као и за приказ резултата програма.

```
void printInstruction(Instruction* instr);

void printInstructions(Instructions* instrs);
```

За рад са променљивама је потребно користити функције из датотеке *Variable.h*. Ту се налазе функције за испис једне или више променљивих у конзолу. Ове функције треба користити приликом откривања грешака у програму, као и за приказ резултата програма.

```
void printVariables(Variables* variables);

void printVariable(Variable* variable);
```

Пример међукода за који се после ради анализа животног века променљивих, те додела ресурса, може се направити позивом функције `makeExample` која враћа листу припремљених инструкција.

Реализација анализе животног века променљивих налази се библиотеци *libLivnessAnalysis.lib* и датотеци *LivnessAnalissysLib.h*. Позивом функције `doLivenessAnalysis` за прослеђену листу инструкција се добија нова листа инструкција са анализом животног века променљивих. Уколико се преко аргумента `debug` пренесе вредност `__DEBUG__` исписаће се додатне информације везане за анализу животног века променљивих.

```
Instructions* doLivenessAnalysis(Instructions* instructions, int debug = __NO_DEBUG__);
```

За рад са графом сметњи је потребно користити функције из датотеке *InterferenceGraph.h*. Помоћу функције `doInterferenceGraph` од листе инструкција се прави граф сметњи, тј. структура `InterferenceGraph` (прва фаза доделе ресурса). Том приликом се заузима меморија за двоструки низ (матрица сметњи) коју је потребно ослободити на крају програма позивом функције `freeInterferenceGraph`. За испис графа сметњи у конзолу треба користити функцију `printInterferenceGraph`.

```
InterferenceGraph* doInterferenceGraph(Instructions* instructions);

void freeInterferenceGraph(InterferenceGraph* ig);

void printInterferenceGraph(InterferenceGraph* ig);
```

Фаза упрошћавања (друга фаза доделе ресурса) је реализована у библиотеци *libResourceAllocation.lib*, односно датотеци *Simplification.h*, у функцији `doSimplification`. Она као аргументе прихвата граф сметњи и степен упрошћавања. Као повратна вредност функције, враћа се попуњен стек (Слика 2). Уколико је показивач на стек празан, тада фаза упрошћавања није успела да упрости граф сметњи за жељени степен, тј. дошло је до преливања (трећа фаза доделе ресурса).

```
stack<Variable*>* doSimplification(InterferenceGraph* ig, int degree);
```

ЗАДАТАК

1. Реализовати фазу избора. Фаза избора налази се у функцији `doResourceAllocation` у датотеци *ResourceAllocation.h* Она као аргументе очекује стек из фазе упрошћавања и показивач на граф сметњи. Уколико се успешно доделе ресурси свим променљивама, ова функција враћа тачну вредност и обрнуто.

```
bool doResourceAllocation(stack<Variable*>* simplificationStack,
InterferenceGraph* ig);
```

ДОДАТНИ ЗАДАТАК

1. Реализовати брисање T_MOVE инструкција у оквиру функције `removeMove` у датотеци *ResourceAllocation.cpp.* Ова функција, након завршетка доделе ресурса, елиминише све инструкције које представљају доделу регистра самом себи.

```
Instructions * removeMove(Instructions * instrs);
```