

Compilers

Fall 2009

Register Allocation

Sample Exercises and Solutions

Prof. Pedro Diniz

Departamento de Engenharia Informática (DEI)
Instituto Superior Técnico / Tagus Park
Av. Prof. Dr. Cavaco Silva
2780-990 Porto Salvo
Portugal

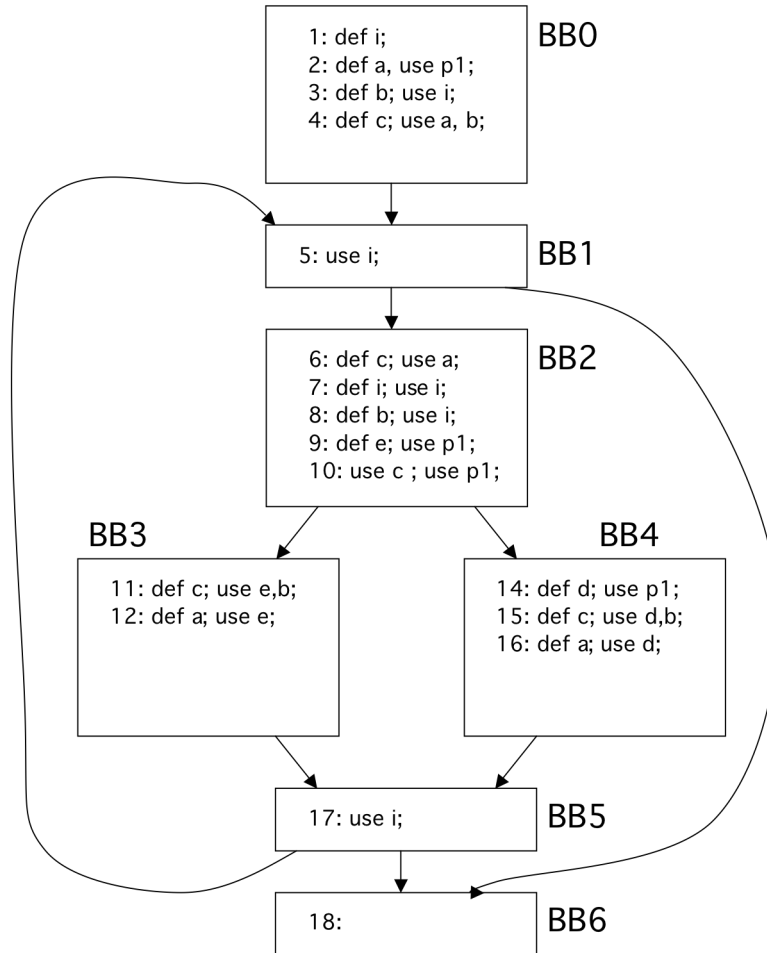
Problem 1: Consider the three-address code below:

```
1:   i = 0;
2:   a = p1;
3:   b = i * 4;
4:   c = a + b;
5: L1: if (i > 100) goto L2
6:   c = a + 1;
7:   i = i + 1;
8:   b = i * 4;
9:   e = p1;
10:  if (c <= p1) goto L3
11:  c = e - b;
12:  a = e;
13:  goto L4
14: L3: d = p1;
15:  c = d + b;
16:  a = d;
17: L4: if (i <= 100) goto L1
18: L2: return;
```

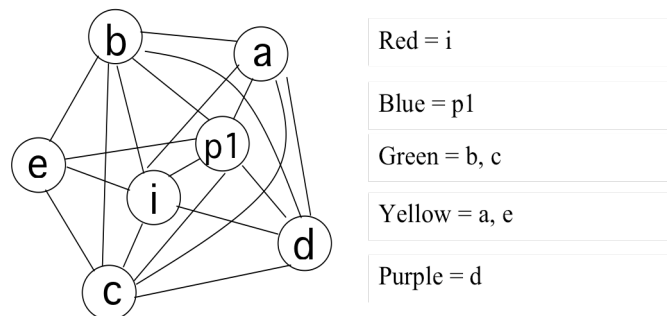
- Determine the interference graph assuming a given variable is live at the end of a specific instruction if after the use in that instruction the value is still used elsewhere in the code. You should assume the variable p1 is defined upon entry of the code as it corresponds to a parameter of the procedure.
- Determine the minimum number of registers needed (without spilling, of course).
- Assuming you were short of one register, which register(s) would you spill and at which points in the program would you insert the spill code? Explain the rationale of your choice.
- Now redo the register allocation using the top-down method in which variables used inside a loop are weighted more than variables outside the loop.

Answer:

- We have drawn the interference webs as shown below where we use the interference notion as dictated by the liveness of a given variable after the execution of a given instruction.



- b) Using these interference webs for computing the interference graph one would get the graph below. To color this graph one needs 4 colors.



- c) There are two cliques of four elements here, i, p1, b and d, and another clique of four with i, p1, c and d so no matter how you would split the live ranges and the corresponding interference webs you could not color the resulting graph with less than 4 colors. You might be able to color the graph with less than 4 colors if you use 2 additional registers for splitting the live ranges corresponding to the variables b and d.
- d) Using the top-down allocation algorithm we must first compute the $COST(V,B)$ functions for each variables V and basic block B. For the example above we have the following matrix of cost values:

	BB0	BB1	BB2	BB3	BB4	BB5	BB6
i	2	1	3	0	0	1	0
p1	0	0	2	1	1	0	0
a	2	0	1	1	1	0	0
b	2	0	1	1	1	0	0
c	1	0	2	1	1	0	0
d	0	0	0	0	3	0	0
e	0	0	1	2	0	0	0

In addition we now have to compute the nesting depth of each basic block as shown in the table below.

	BB0	BB1	BB2	BB3	BB4	BB5	BB6
Depth	0	1	1	1	1	1	0

Given these two tables we now compute the total cost associated with a given variable in this algorithm as $\text{TotCost}(V) = \sum \text{cost}(V, B) * \text{freq}(B)$ where $\text{freq}(B)$ of the basic block B is $10^{\text{depth}(B)}$.

V	TotCost(V)
i	$3 + 1*10 + 3*10 + 1*10 = 53$
p1	$0 + 2*10 + 1*10 + 1*10 = 40$
a	$2 + 1*10 + 1*10 + 1*10 = 32$
b	$2 + 1*10 + 1*10 + 1*10 = 32$
c	$1 + 2*10 + 1*10 + 1*10 = 41$
d	$0 + 3*10 = 30$
e	$1*10 + 2*10 = 30$

As such the variables are assigned register in the following priority: {i, c, a, b, p1, d, e}

Problem 2: Consider the following three-address format representation of a computation. Here we have use the ARP register to load local variables into temporary variables introduced in the intermediate code generation process. We have also used compile-time offsets to load the values of several local variables as they are located and fixed locations in the current procedure AR.

```

01:    t1 = k * 8
02:    t2 = FP + offset_A
03:    t3 = t1 + t2
04:    t4 = *t3
05:    t5 = t4 * x
06:    t6 = FP + offset_B
07:    t7 = t6 * 8
08:    t8 = *t7
09:    t9 = t5 + t8
10:    *t7 = t9
11:    k = k + 1

```

Questions:

- Using the bottom-up register allocator described in class assign the various temporary variables and variables to actual registers. For the purpose of this section, assume you only have 4 physical registers. At each point when choosing to reuse a register indicate why do you pick each one.
- Use the graph-coloring based algorithm for doing register allocation instead. In this section we are to explore the use of interference webs for different definitions of interference as mentioned in class.
 - In the first web use the definition that two variables interfere if there is at least one instruction in which they participate. Derive the interference web between variables using this definition.
 - In the second definition there is no interference if the two webs either do not intersect at all or if they do intersect at a single instruction the web that ends at that instruction participates as the argument of the instruction and the web that begins at that instruction corresponds to the destination value of the instruction.
 - For both definitions determine the minimum number of required registers.
 - Using the graph-coloring heuristic described in class determine the number of required registers for each of the two interference definitions. Why do they differ, or why not?

Answers:

- Below is a description on how to translate the code in this question using 4 physical registers and using the local allocation algorithm described in class. For each variable we have loaded its value into a register via the indirection using the FP and a specific offset in the AR. At specific points in the execution we indicate the current assignment of registers to program variables.

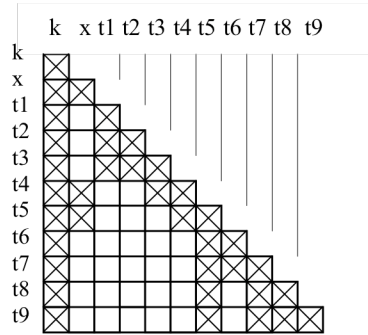
```

01:    r0 = FP + offset_K      // Assignment: r0 ← k; r1 ← empty; r2 ← empty; r3 ← empty;
02:    r1 = r0 * 8             // Assignment: r0 ← k; r1 ← t1; r2 ← empty; r3 ← empty;
03:    r2 = FP + offset_A     // Assignment: r0 ← k; r1 ← t1; r2 ← t2; r3 ← empty;
04:    r3 = r1 + r2           // Assignment: r0 ← k; r1 ← t1; r2 ← t2; r3 ← t3;
05:    r1 = (r3)              // Assignment: r0 ← k; r1 ← t4; r2 ← t2; r3 ← t3;
06:    r2 = FP + offset_X     // Assignment: r0 ← k; r1 ← t4; r2 ← x; r3 ← t3;
07:    r1 = r1 * r2           // Assignment: r0 ← k; r1 ← t5; r2 ← x; r3 ← t3;
08:    r2 = FP + offset_B     // Assignment: r0 ← k; r1 ← t5; r2 ← t6; r3 ← t7;
09:    r3 = r2 * 8            // Assignment: r0 ← k; r1 ← t5; r2 ← t6; r3 ← t7;
10:    r2 = *r3               // Assignment: r0 ← k; r1 ← t5; r2 ← t8; r3 ← t7;
11:    r1 = r2 + r1           // Assignment: r0 ← k; r1 ← t9; r2 ← t8; r3 ← t7;

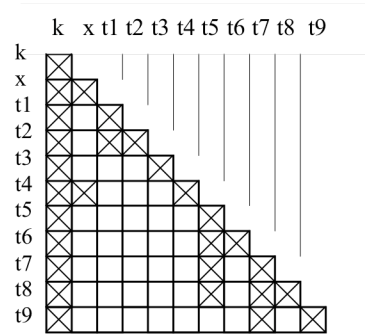
```

12: $*r3 = r1$ // Assignment: $r0 \leftarrow k; r1 \leftarrow t9; r2 \leftarrow t8; r3 \leftarrow t7$;
 13: $r0 = r0 + 1$ // Assignment: $r0 \leftarrow k; r1 \leftarrow t9; r2 \leftarrow t8; r3 \leftarrow t7$;

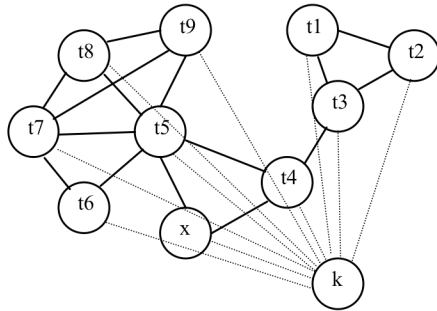
- (b) We present the interference table instead as it is easier to read. On the left we have the interference table for the simpler notion of interference whereas on the right we have for the notion of interference taking into account the use of the value in each register when the RHS is evaluated.



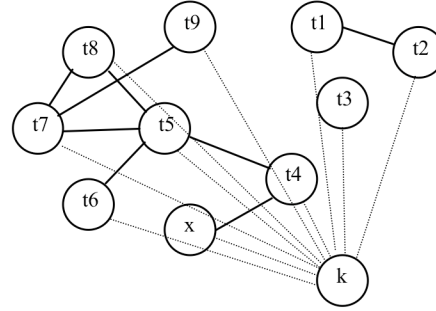
a. Interference Table



b. Interference Table

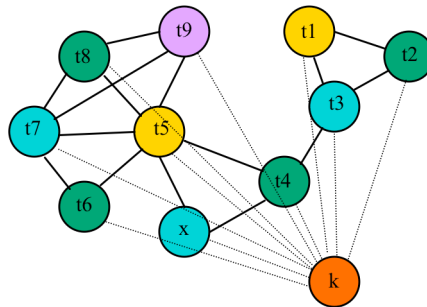


a. Interference Graph



b. Interference Graph

- (c) In the first case we need 5 registers as the maximum number of conflicts in any row of the table is 4 (last one for t9) whereas in the second case we only need 4 registers.
- (d) Using the graph-coloring heuristic algorithm (for the first notion of interference) and using 5 registers during the first phase we push the nodes for all nodes (in lexicographical order) onto the stack leaving only the nodes corresponding to t5 and k. We assign r0 to k and r1 to t5. Then popping each of the nodes off the stack we can assign r1 to t1, r2 to t2, t4, t6 and t8; r3 to t3, x, t7 and r4 to t9, yielding the colored graph below:



Problem 3:

```

1:    i = 0
2:    a = 1
3:    b = i * 4
4:    c = a + b
5: L1: if i > n goto L2
6:    c = a + 1
7:    i = i + 1
8:    b = i * 4
9:    if c <= p goto L3
10:   e = 1
11:   c = e - b
12:   a = e
13:   goto L4
14: L3: d = 2
15:   c = d + b
16:   a = d
17: L4: goto L1
18: L2: return

```

For the code shown above, determine the following:

- The live ranges for the variables i , a , b , c , d , e , p .
- The interference graph using the simplest definition of interference where the webs include the live range in terms of the line numbers where an access to a variable (either a read or a write operation) occurs.
- Determine the minimum number of registers needed (without spilling, of course) using the graph coloring algorithm described in class.
- Assuming that you were short of one register, which register(s) would you spill and at which points in the program would you insert the spill code? Explain.

Answers:

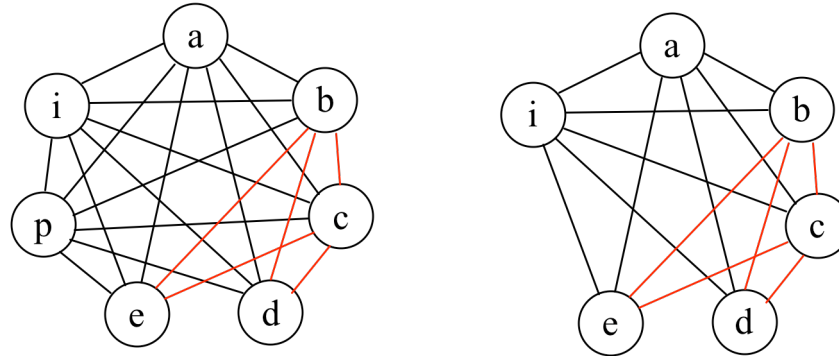
- The basic observation is that one variable is live at a specific program point p if its value is still going to possibly be used in the future. The best way to figure this out is to begin by constructing the control-flow graph (CFG) for this code. For instance variable i is live at program point 11 as there is a possibility that the value defined for this variable at point 7 is going to be used along the path that traverses the basic block that contains the instructions 10 through 13 which then jumps to line 5 which reads this i variable to evaluate the test ($i > n$). Tracing the definitions and uses for each variable we can arrive at the following live ranges where the number indication the source program line numbers:

```

i: {1 – 17}
p: {1-17}
a: {1-17}
b: {3,4}, {8,9,10,11}, {8,9,14,15}
c: {4}, {6}, {11}, {16}
d: {14,15,16}
e: {10,11,12}

```

- b) The variables $\{i, a, p\}$ interfere with all the other variables (using block edges). The complete interference graph is as shown below on the left.



- c) Using the algorithm we have studied in class for graph coloring for $N = 6$ given that that is the highest degree in the graph it is highly likely that for $N = 6$ we will be able to find such coloring of the interference graph. In this case only nodes d and e go into the stack as they have degree 5 less than 6. If that is the case we now remove one node from the graph, say node p that interferes with all the other remaining nodes in the graph. We are left with the graph above in the middle. Then we must push all the remaining nodes on the stack again, this time all of them have degree less than 6. We then pop the nodes one by one and assign colors. One such color is to assign the same color, say red to $\{e, d\}$ as they do not interfere and any other 4 colors to the remaining nodes. This results in 5 colors leaving the node corresponding to p uncolored.
- d) If you had one less register say 4 registers, you could leave the node with the shortest live range, say node d or e out of registers.

Problem 4: Consider the following three-address format representation of a computation using scalars and arrays A and B. Assume for the purposes of this exercise that no additional registers are needed to access (either writing or reading) the value of an array. As such the access to $A[t1]$ requires no additional registers in addition to the one carrying the value of $t1$.

```

01:  t1 = i
02:  t2 = A[t1]
03:  t3 = i + 1
04:  t4 = A[t3]
05:  t5 = t2 + t4
06:  t6 = t5 / 2
07:  t7 = i
08:  B[t7] = t6
09:  i = i + 1

```

Questions:

- (c) Using the bottom-up register allocator described in class assign the various temporary variables and variables to actual registers. For the purpose of this section, assume you have 3 physical registers and that the scalar i is already loaded into register $r0$. At each point when choosing to reuse a register indicate why do you pick each one. Also you should try to reuse registers in copy operations such as $t1 = i$ in line 01 thus not consuming any more registers but simply propagating the use of $r0$ in this case.
- (d) Use the graph-coloring based algorithm for doing register allocation instead. In this section we are to explore the use of interference webs for different definitions of interference as mentioned in class.
 - a. In the first web use the definition that two variables interfere if there is at least one instruction in which they participate. Derive the interference web between the variables in this code using this definition.
 - b. In the second definition there is no interference if the two webs either do not intersect at all or if they do intersect at a single instruction the web that ends at that instruction participates as the argument of the instruction and the web that begins at that instruction corresponds to the destination value of the instruction.
 - c. Using the graph-coloring heuristic described in class with $N=4$ and determine the coloring and hence register allocation for both interference definitions. Why do they differ, or why not?

Answers:

- (a) Below is a description on how to translate the code in this question using 3 physical registers and using the local allocation algorithm described in class. At specific points in the execution we indicate the current assignment of registers to program variables. Note that we could have eliminated $t1$ as the first instruction makes a copy of the value of i into $t1$ (and the same is true for $t7$ and i). For this reason in the assignment below the pair $t1,i$ and $t7,i$ appear as a single tuple.

01:	$r0 = r0$	// Assignment: $r0 \leftarrow t1, i;$	$r1 \leftarrow \text{empty};$	$r2 \leftarrow \text{empty};$
02:	$r1 = A[r0]$	// Assignment: $r0 \leftarrow t1, i;$	$r1 \leftarrow t2;$	$r2 \leftarrow \text{empty};$
03:	$r2 = r0 + 1$	// Assignment: $r0 \leftarrow t1, i;$	$r1 \leftarrow t2;$	$r2 \leftarrow t3;$
04:	$r0 = A[r2]$	// Assignment: $r0 \leftarrow t4;$	$r1 \leftarrow t2;$	$r2 \leftarrow t3;$
05:	$r2 = r1 + r0$	// Assignment: $r0 \leftarrow t4;$	$r1 \leftarrow t2;$	$r2 \leftarrow t5;$
06:	$r0 = r2 / 2$	// Assignment: $r0 \leftarrow t6;$	$r1 \leftarrow t2;$	$r2 \leftarrow t5;$
07:	$r1 = i$	// Assignment: $r0 \leftarrow t6;$	$r1 \leftarrow t7, i;$	$r2 \leftarrow t5;$
08:	$B[r1] = r0$	// Assignment: $r0 \leftarrow t6;$	$r1 \leftarrow t7, i;$	$r2 \leftarrow t5;$
09:	$r1 = r1 + 1$	// Assignment: $r0 \leftarrow t6;$	$r1 \leftarrow t7, i;$	$r2 \leftarrow t5;$

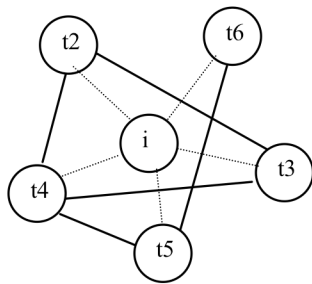
- (e) We present the interference table instead as it is easier to read. On the left we have the interference table for the simpler notion of interference whereas on the right we have for the notion of interference taking into account the use of the value in each register when the RHS is evaluated.

	i	t2	t3	t4	t5	t6
i	X					
t2		X				
t3		X	X			
t4		X	X	X		
t5		X	X	X	X	
t6						X

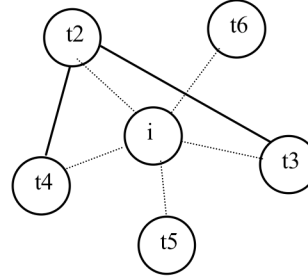
a. Interference Table

	i	t2	t3	t4	t5	t6
i	X					
t2		X				
t3		X	X			
t4		X	X	X		
t5		X	X	X	X	
t6						X

b. Interference Table

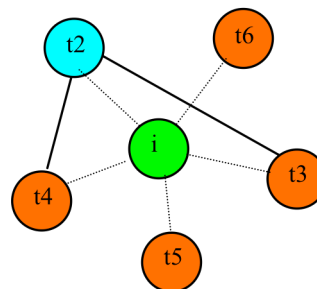
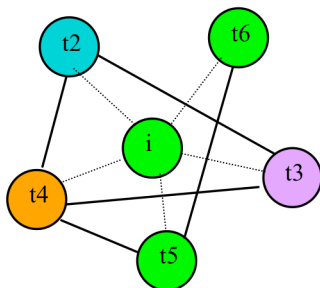


a. Interference Graph



b. Interference Graph

- (f) In the first case we need 4 registers as the maximum number of conflicts in any row of the table is 4 (for t2) whereas in the second case we only need 3 registers. There are several cliques of size 3 in the second case and at least one clique of size 4 in the first case (involving i).
- (g) Using the graph-coloring heuristic algorithm (for the first notion of interference) and using 4 registers or $N=4$ in the algorithm, we first push onto the stack the nodes in the order $\{6, 5, 4, 3, 2, i\}$. We then pop the node corresponding to i and assign it the color c_0 . Next we color t_2 with color c_1 and color t_3 with color c_2 . Next we color t_4 with color c_3 when we take it off the stack and when we pop t_5 we can color it with c_1 . As for t_6 we can color it with color c_1 as well as it does not interfere with any other node in the graph. The figures below depict these colorings on the left for the first interference graph and on the right for the second interference graph which uses only 3 color.



Problem 5: Consider the following three-address format representation of a computation using scalars and arrays A and B and a procedure $F(\text{int } a, \text{int } b)$. Assume for the purposes of this exercise that no additional registers are needed to access (either writing or reading) the value of an array. As such the access to $A[t1]$ requires no additional registers in addition to the one carrying the value of $t1$.

```

01:  t1 = i
02:  t2 = A[t1]
03:  t3 = i + 1
04:  t4 = A[t3]
05:  t5 = t2 + t4
06:  t6 = t5 / 2
07:  t7 = i
08:  B[t7] = t6
09:  putparam i
10:  putparam 1
11:  call F, 2
12:  t8 = i
13:  t9 = t8 + 2

```

- Apply copy propagation and dead code elimination to the basic block immediately preceding the function invocation.
- Use the bottom-up register allocation algorithm for 3 registers under the assumption that variable i is live outside this basic block and after the call to F . In this section you can simply ignore the instructions on lines 09 through 11. In this part of the code load the value of the scalar variables using the offset of the scalar from the Frame-Pointer (FP) register. So you need to make some changes to the intermediated code.
- Passing the arguments to the procedure F could be done via registers. This would come at the cost of additional registers unless the values required by a given procedure would already reside in registers. For this particular case explain which `putparam` instruction could you eliminate by using arguments for the procedure that are already in registers.

Answers:

- The statements on lines 01, 07 and 12 are dead once we propagate the corresponding assignments to the instructions on lines 02, 08 and 13.

```

01:  t1 = i
02:  t2 = A[i]
03:  t3 = i + 1
04:  t4 = A[t3]
05:  t5 = t2 + t4
06:  t6 = t5 / 2
07:  t7 = i
08:  B[i] = t6
09:  putparam i
10:  putparam 1
11:  call F, 2
12:  t8 = i
13:  t9 = t8 + 2

```

- b) The code below illustrates the resulting code under the assumption that the variable `i` is initially stored at an negative offset 16 of the Frame-Pointer (FP) register.

```

01:  r0 = FP - 16
02:  r0 = *r0          // loads the value of variable i into r0
02:  r1 = A[r0]        // r0 ← i,t1,  r1 ← t2,  r2 ← empty
03:  r2 = r0 + 1       // r0 ← i,t1,  r1 ← t2,  r2 ← t3
04:  r0 = A[r2]        // r0 ← t4,   r1 ← t2,  r2 ← t3
05:  r2 = r1 + r0      // r0 ← t4,   r1 ← t2,  r2 ← t5
06:  r0 = r2 / 2       // r0 ← t6,   r1 ← t2,  r2 ← t5
07:  r1 = FP - 16     // r0 ← t6,   r1 ← i,t7, r2 ← t5
08:  r1 = *r1         // r0 ← t6,   r1 ← i,t7, r2 ← t5
08:  B[r1] = r0        // r0 ← t6,   r1 ← i,t7, r2 ← t5
09:  putparam r1
10:  putparam 1
11:  call F,2
12:  r0 = r1           // r0 ← t8,   r1 ← i,t7, r2 ← t5
13:  r2 = r1 + 2       // r0 ← t8,   r1 ← i,t7, r2 ← t9

```

- c) For this particular case the argument value `i` is already in register `r1`, so we could eliminate the first `putparam` instruction provided now that when doing the register allocation for `F` we would start with the assumption that the first parameter is already in `r1`.

Problem 6: The code shown on the left is for a procedure in C where p and n are arguments to the function.

```

1 func:getparam p
2   getparam n
3   i = 0
4   a = 1
5   b = i * 4
6   c = a + b
7 L1:  if i > n goto L2
8     c = a + 1
9     i = i + 1
10    b = i * 4
11    if c <= p goto L3
12    e = 1
13    c = e - b
14    a = e
15    goto L4
16 L3: d = 2
17    c = d + b
18    a = d
19 L4: goto L1
20 L2: return

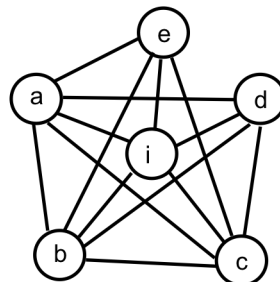
```

For this code determine the following:

- The live ranges for the variables i , a , b , c , d , e .
- The interference graph using the simplest definition of interference where the webs include the live range in terms of the line numbers where an access to a variable (either a read or a write operation) occurs. Assume that the parameters n and p are not in registers.
- Determine which variables should be put in registers using the top-down register allocation algorithm described in class given that you only had 4 registers. Explain.

Answers:

- Variables i is live at all instructions in this procedures and thus its live ranges are $\{3, \dots, 20\}$. Variable a has two live ranges $r1 = \{4, 5, 6, 7, 8, 9, 10, 12, 12, 13, 14\}$ and $r2 = \{18, 19, 7, 8\}$ in this second range the value defined in line 18 is used in instruction 8 via the goto statement on line 19. For variable b we have three live ranges, $r1 = \{5, 6\}$, $r2 = \{10, 11, 12, 13\}$ and $r3 = \{10, 11, 16, 17\}$. For variable c we have 4 live ranges $r1 = \{6\}$, $r2 = \{8, 9, 10, 11\}$, $r3 = \{13\}$ and $r4 = \{17\}$. For variable d we have a single live range $r1 = \{16, 17, 18\}$ and for e we have $r1 = \{12, 13, 14\}$.
- The interference graph using the simplest definition of interference where the webs include the live range in terms of the line numbers where an access to a variable (either a read or a write operation) occurs. Assume that the parameters n and p are not in registers. Using the information from a) the interference graph is as shown below. The only two variables that do not interfere (using this definition) with each other are variables d and e .



- c) Determine which variables should be put in registers using the top-down register allocation algorithm described in class given that you only had 5 registers. Explain. We need to determine the loop structure of the code so that we can weight the occurrences of each of the variables. By inspection of the code we can see that there is a loop that contains all statements from lines 7 through 19. Inside the loop there is an if-then-else construct so all the accesses inside this loop will be weighted with weight 10 as there is a single nested loop here.

As such the table of weights for this code is as follows:

	Variable					
	i	a	b	c	d	e
Breakdown	$4 \times 10 + 2$	$30 \times 10 + 2$	$3 \times 10 + 2$	$4 \times 10 + 1$	$3 \times 10 + 0$	$3 \times 10 + 0$
Weight	42	32	32	41	30	30

We can see that preferentially we should put in registers the variables *i* and *c* followed by the variables *a* and *b* and finally the variables *d* and *e*. Note that because variables *d* and *e* do not interfere (see b) above) they need only one register and thus I would be able to allocate all of them to the 5 registers provided.