Exercise 10.1 Colorability

Show that Chaitin’s algorithm finds a $k$-coloring for every $k$-colorable chordal graph.

*Hint:* Every chordal graph with at least 2 nodes has 2 simplicial nodes.

**Solution:** If the chordal graph consists of only one node then it is colorable with $k$ colors for any $k \geq 1$. As we do not have self edges in our interference graphs by definition, the degree of the only node is 0. So the reduce phase of the algorithm can always remove this one node and end up with the empty graph since $0 < k$ for all $k \geq 1$.

If the chordal graph consists of two or more nodes, then we may assume that it contains at least two simplicial nodes. Each of the simplicial nodes forms a clique with all its neighbors by the definition. Since the graph is by assumption $k$-colorable, this means that the clique is at most of size $k$. As a consequence each of this simplicial nodes has at most $k - 1$ neighbors. This means the local colorability criterion of Chaitin finds always a node to remove of the chordal graph. Note that the node actually removed does not have to be one of the simplicial nodes, but the simplicial node are always a last resort if there is no other node to remove.

Next we have to show that we will always end up again in another chordal graph that is $k$-colorable after having removed the node selected by Chaitin’s criterion for all graphs. By definition a chordal graph contains no induced cycle of a length greater than 3. This means especially that the induced subgraph created by removing the node selected by Chaitin’s criterion does also not contain any induced cycles of length greater than 3 since otherwise also the original graph would contain those induced cycles of length greater than 3. This again means that the reduced graph is again a chordal graph. Since the original graph was $k$-colorable and we did not add any edges to it when we removed the node selected, we can be sure that the resulting graph is also $k$-colorable.

We know that when we start with a chordal graph we will always find a node to remove and the resulting graph will always be chordal again. So starting with a chordal graph, we will always end up with the empty graph after applying the reduce phase of Chaitin’s algorithm.

As soon as we have reached the empty graph, we can always find a $k$-coloring using Chaitin’s algorithm since we insert the nodes in the reverse order we removed them. Therefore any node inserted will at most have $k - 1$ neighbors in the graph directly after insertion. So we have at least one of our $k$ colors left to color the node just inserted.

Exercise 10.2 Partitioned Boolean Quadratic Programming

Prove that finding a solution for a PBQP problem to be NP-hard by reducing SAT to PBQP.

*Hint:* Reconsider the NP-hardness proof for instruction selection. First, try to map the boolean formula $(a \land b) \lor \neg b$ from the example in Figure 2 of Koes’ paper to PBQP. Then, you can derive an algorithm to map any SAT problem to PBQP. Generally, to map $a \lor b$ you will need four nodes: one for $a$, one for $b$, one for $\lor$ and an auxiliary node.

**Solution:**

To prove that PBQP is NP-hard, we given a polynomial time algorithm that, given a boolean formula $\Phi$, produces a PBQP instance that has a feasible solution if and only if there is a satisfying assignment for $\Phi$. We will describe the PBQP instance in the form of a graph with cost vectors attached to nodes and cost matrices attached to edges as we have seen it in the lecture.

We start with a graph with a node for each sub-expression of $\Phi$. To the node of $\Phi$ itself, we attach the two-dimensional cost vector $(0, \infty)^T$, all other nodes get the cost vector $(0, 0)^T$.

The idea is that a solution to the PBQP instance corresponds to a satisfying valuation of $\Phi$. If the PBQP solution chooses the first component for such a node, it should mean that the corresponding subexpression evaluates to $true$ in the valuation, whereas choosing the second component means that it evaluates to $false$. By setting the second component of the cost vector of the node that corresponds to $\Phi$ to $\infty$, we make sure that we only allow valuations that make the entire formula evaluate to $true$, i.e. that satisfy $\Phi$.

---

1 available at [https://dl.acm.org/citation.cfm?id=1356065](https://dl.acm.org/citation.cfm?id=1356065) (from university network)
We now have to connect these nodes in a way that a PBQP solver can only choose components for nodes which correspond to non-atomic expressions that are allowed in this interpretation with respect to their operator and their subexpressions. We do that as follows:

- For a node representing an expression of the form $\neg e$, we insert the following connection:

![Diagram of ¬e connection]

- For a node representing an expression of the form $e_1 \land e_2$, we insert the following connection with a fresh node $t$:

![Diagram of e1 ∧ e2 connection]

The connections between $t$ and $e_1$ and $e_2$ make sure that for each selection of components for $e_1$ and $e_2$, exactly one component of $t$ can be selected in a feasible solution (the first one if both are false, the second one if $e_1$ is false and $e_2$ is true, the third one if $e_1$ is true and $e_2$ is false, and the fourth one if both are true). The connection between $t$ and $e_1 \land e_2$ encodes the value table of $\land$: If the fourth component of $t$ is chosen (i.e. both operands are true), only the first component of the result node can be selected (i.e. it is true). Otherwise, only the second component of the result node can be selected (i.e. it is false).

- Following the same pattern, for a node representing an expression of the form $e_1 \lor e_2$, we insert the following connection with a fresh node $t$:
This construction ensures that each solution of the PBQP instance corresponds to a valuation of $\Phi$. We can extract the values of a variable from a solution by checking which component of the corresponding node is selected.

In order for this to be a proof of NP-hardness of PBQP, we have to argue that this construction is possible to perform in polynomial time. The argument works as follows: In the first step, we have to insert one node for each subexpression of $\Phi$, these are linearly many node insertions with respect to the size of $\Phi$. In the second step, we have to insert for each non-atomic subexpression either one edge or three edges and an additional node. These are again linearly many insertions with respect to the size of $\Phi$. Assuming that we can represent a graph annotated with vectors and matrices with at most 8 entries with time requirements for insertions that are polynomial in the sizes of the graph and the insertee, we can perform the construction in polynomial time.

**Exercise 10.3 PBQP Applied**

1. Study the LLVM-IR program below and draw the value graph for the loop body (for.body). Include constants, function arguments and PHI nodes from other blocks in the graph. Futhermore, replace the getelementptr instruction by appropriate scalar operations (add/mul) and fold constant expressions together. Assume the size of an i32 is 4 bytes.

2. Use the patterns on the PBQP slide 19 and the costs shown below to create a PBQP instance for the graph constructed in part 1. Assume the patterns $AC$ and $A$ are also available for multiplications ($MC/M$) and that there is a pattern $AR$ for accessing an argument.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>AR</th>
<th>C</th>
<th>P</th>
<th>A</th>
<th>AC</th>
<th>M</th>
<th>MC</th>
<th>L</th>
<th>LA</th>
<th>LAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>40</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

3. Use the optimality-preserving reductions and the heuristic reduction to find a solution for the PBQP problem. Write down the order in which edges/nodes are eliminated and the rule that was applied.

```llc
1 define i32 @array_sum(i32* %A, i32* B, i32 %N) {
2 entry:
3 br label %for.cond
4
5 for.cond:
6 %iv = phi i32 [ 0, %entry ], [ %iv.inc, %for.body ]
7 %sum = phi i32 [ 0, %entry ], [ %add1, %for.body ]
8 %B.cur = phi i32* [ %B, %entry ], [ %B.idx, %for.body ]
9 %cmp = icmp slt i32 %iv, %N
10 br i1 %cmp, label %for.body, label %for.end
11
12 for.body:
13 %A.idx = getelementptr i32, i32* A, i32 %iv
14 %B.idx = getelementptr i32, i32* B.cur, i32 1
15 %A.val = load i32, i32* A.idx, align 4
```
Solution:

1. We find the following value graph:

2. We attach the possible matches to the nodes:
Next, we add matrices to the edges:

To obtain our PBQP problem, we insert the actual costs for the patterns:
3. We start by cutting independent edges and obtain:

Next, we apply the rule for nodes of degree 1 where applicable:
Next, we again apply the rule for nodes of degree 1 where applicable:

Now, all nodes are separated, so we can choose for each node the locally minimal option. This gives us a solution with cost 370, it looks as follows:
Since we did not use the heuristic function, the result is actually an optimal instruction selection.

Project task G  Implement Sparse Conditional Constant Propagation

For this project assignment, we introduce an additional compiler switch, --optimize. This switch performs the same operation as --compile and additionally performs optimizations. Implement the following optimization:

- Perform the SCCP \(^2\) analysis.
- If applicable,
  - substitute instructions with constants,
  - substitute conditional branches with unconditional ones,
  - remove dead instructions and
  - remove unreachable blocks.
- If you do not want to implement SSA construction yourself, you may use the LLVM \texttt{mem2reg} pass (see \texttt{llvm::createPromoteMemoryToRegisterPass()}) \textbf{but no other LLVM passes}.
- This is the last project assignment. Information on scheduling your code review meeting with us will be available in the forum.
- The soft deadline for this milestone is 2018-02-02.
- Keep it simple!

\(^2\)consider https://dl.acm.org/citation.cfm?id=103136 for more details if you haven’t already (accessible from the university network).