An implementation of chordal graph algorithms in FGL

William Maxwell

Overview

Chordal graphs

Chordal graphs in FGL

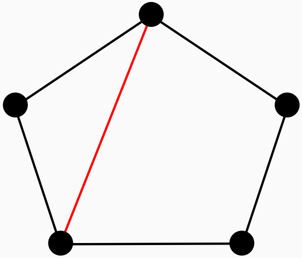
Algorithm implementations

Tree decompositions

Chordal graphs

Chordal graphs

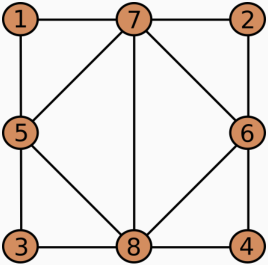
Definition
A graph is called a chordal graph if every cycle in the graph has a chord.



The red edge is a chord, but the graph is not chordal

Chordal graphs

Definition
A graph is called chordal if the vertices can be ordered $v_1, v_2, \ldots, v_n$ such that for each $1 \le i \le n$, v_i and its neighbors with index greater than i form a clique.



A chordal graph and its ordering

Algorithmic results

Many NP-Complete problems are polynomial or linear on chordal graphs. A few well known problems include

Maximum clique
Maximum coloring
Maximum independent set
Minimum width tree decomposition

Chordal graphs in FGL

Graph data types

```
type Node         = Int
type Adj b        = [(b, Node)]
type Context a b = (Adj b, Node, a, Adj b)

data Graph a b   = Empty | Context a b & Graph a b
```

In FGL `Graph` is actually a type class and the constructors are functions required by the implementation of the type class

Algorithm implementations

Maximum clique

```haskell
-- Compute the size of a maximum clique
maxClique :: Gr a b -> Int
maxClique g = 1 + ufold (\c k -> max (length $ suc' c) k) 0 g

-- Compute a maximum clique
maxClique :: Gr a b -> [Node]
maxClique = ufold maxClique' []

maxClique' :: Context a b -> [Node] -> [Node]
maxClique' c ns = if length ns' > length ns then ns' else ns
    where ns' = n : suc' c
          n   = node' c
```

Chordal completion

```haskell
chordalCompletion :: Gr a () -> Gr a ()
chordalCompletion g = ufold chordalCompletion' g g

chordalCompletion' :: Context a () -> Gr a () -> Gr a ()
chordalCompletion' c g = addClique g (map fst $ suc' c)


addClique :: Gr a () -> [Node] -> Gr a ()
addClique g []     = g
addClique g (n:ns) = addClique gc ns
    where gc = insEdges fs g
          es = labEdges g
          fs = [x | x <- zip3 (repeat n) ns (repeat ()), not (x `elem` es)]
```

Is Chordal?

```haskell
isChordal :: Gr a b -> Bool
isChordal g = isChordal' g (edges g)

isChordal' :: Gr a b -> [Edge] -> Bool
isChordal' g es
    | isEmpty g = True
    | otherwise = isClique sucs es && isChordal' g' es
      where (c,g') = matchAny g
            sucs   = suc' c

isClique :: [Node] -> [Edge] -> Bool
isClique [] _ = True
isClique (n:ns) es = checkVertex n ns && isClique ns es
    where checkVertex n [] = True
          checkVertex n (n':ns) = (n,n') `elem` es && checkVertex n ns
```

Tree decompositions

Tree decompositions

A tree decomposition of a graph G is a tree whose vertices (called bags) are sets of vertices from G satisfying some a few properties

A tree decomposition tells us how "treelike" our graph is

Removing a single bag from G disconnects G similar to how removing a single vertex from a tree disconnects the tree

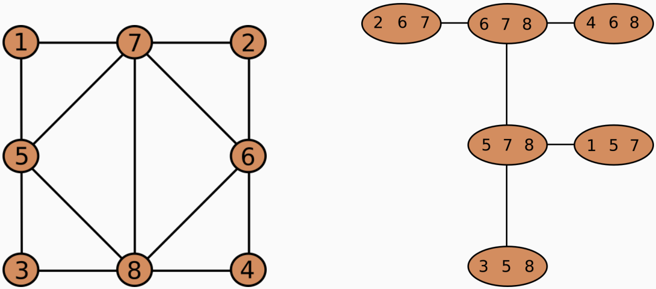
The size of the largest bag determines the treewidth of a graph

Tree decompositions

Many NP-Complete problems are efficient on graphs with small treewidth

Problems are typically solved via dynamic programming on the tree decomposition

An example



A graph and a tree decomposition

Tree decomposition types

```
type Bag        = Set Node
type TreeDecomp = Gr Bag ()
```

Imperative tree decomposition algorithm

Algorithm 2 Construct a TD T_π of a graph G using elimination ordering π and Gavril's algorithm

INPUT: Graph $G = (V, E)$, π a permutation of V

OUTPUT: TD $T_\pi = (X, (I, F))$ with (I, F) a tree, and bags $X = \{X_i\}$, $X_i \subseteq V$

Initialize $T = (X, (I, F))$ with $X = I = F = \emptyset$, $n = |V|$

Create an empty n-long array $t[]$

Use Algorithm 1 to create a triangulation G_π^+ using π.

Let $k = 1$, $I = \{1\}$, $X_1 = \{\pi_n\}$, $t[\pi_n] = 1$

for $i = n - 1$ to 1 **do**

    Find $B_i = \{\text{neighbors of } \pi_i \text{ in } G_\pi^+\} \cap \{\pi_{i+1}, \ldots, \pi_n\}$

    Find $m = j$ such that $j \leq k$ for all $\pi_k \in B_i$

    if $B_i = X_{t[m]}$ **then**

        $X_{t[m]} = X_{t[m]} \cup \{\pi_i\}$; $t[\pi_i] = t[m]$

    else

        $k = k + 1$

        $I = I \cup \{k\}$; $X_k = B_i \cup \{\pi_i\}$

        $F = F \cup \{(k, t[m])\}$; $t[\pi_i] = k$

    end if

end for

return $T_\pi = (X, (I, F))$

Aaron B. Adcock and Blair D. Sullivan and Michael W. Mahoney, Tree decompositions and social graphs

Tree decomposition types

```haskell
type SucSet      = Node -> Set Node

data TDState = TDState {
    bagMap    :: IntMap Node,
    sucMap    :: SucSet,
    partialTD :: TreeDecomp,
    currBag   :: Node
}

buildTreeDecomp :: Gr a b -> State TDState ()
```

The SucSet type

```haskell
addSucs :: SucSet -> Node -> [Node] -> SucSet
addSucs f n ns = \x -> if x `elem` ns then Set.insert n (f x) else f x

getSucSet :: Gr a b -> SucSet
getSucSet = ufold (\c f -> addSucs f (node' c) (pre' c)) (const Set.empty)
```

Adding a vertex to a bag

```haskell
-- First node is the bag, second node is the new vertex
addToBag :: TreeDecomp -> Node -> Node -> TreeDecomp
addToBag td b v = insNode (b, lb') (delNode b td)
    where lb  = case lab td b of (Just l) -> l
                                 Nothing  -> error "Bag not found"
          lb' = Set.insert v lb
```