Factors

P. Danziger

1 Factors and Factorizations

Definition 1

1. *A factor is a spanning subgraph, H of a graph G.*
   Note that $V(H) = V(G)$ and $E(H) \subseteq V(G)$.

2. *A $k-$factor is a spanning subgraph H of G in which every member of H has degree k.*

3. *A $k-$factorization of a graph G is a set of $k-$factors $H_1, \ldots H_\ell$ such that $G = H_1 \cup \ldots H_\ell$ and $E(H_i) \cap E(H_j) = \emptyset$. So every edge of G is in exactly one of the $k-$factors $H_1, \ldots, H_k$.*

Notes

1. A $1-$factor almost the same as a perfect matching. The subtle difference is that a perfect matching is a collection of edges, but a $1-$factor is a graph.

2. A $1-$factorization is a partition of the edge set of G into $1-$factors (perfect matchings).

3. A $2-$factor is a collection of cycles.

   Note The sizes of the cycles may vary in each factor, and across factors.

   A special case is when the size of the $2-$factor is n - the number of vertices in G. In this case a $2-$factor is a Hamiltonian cycle.

Theorem 2 *In order for a graph to have a $\ell-$factorization it must be $k-$regular where ℓ divides k.*

Proof: Each vertex must appear in every $\ell-$factor. Each time it appears in a $\ell-$factor it uses j of its incident edges.
Note that there are $k-$regular graphs which do not have a $\ell-$factorization. i.e. this condition is necessary, but not sufficient. For example the Peterson graph does not have a 1-factorization.

Theorem 3 *Every $k-$regular bipartite graph has a $1-$factorization.*

Theorem 4 *A graph G has a $2-$factorization if and only if G is $k-$regular for some even k.*

Proof: ($\Rightarrow$) Necessity follows from Theorem 2.
($\Leftarrow$) Let G be a $2k-$regular graph, with $V(G) = \{v_1, \ldots v_n\}$.
Since every vertex has even degree G has an Eulerian cycle C.
This cycle uses every edge, but may visit vertices more than once.
We now use C to construct a new bipartite graph H as follows:
The parts of H are $X = \{u_1 \ldots u_n\}$ and $Y = \{v_1 \ldots v_n\}$.
There is an edge between u_i to w_j if and only if v_i and v_j are adjacent in the Eulerian cycle C.

Now H is $k-$regular since each point appears k times in the cycle C.
Thus H is bipartite and $k-$regular and so has a $1-$factorization by Theorem 3.
Let $\{F_1, \ldots, F_r\}$ be the $1-$factorization.
Consider F_1, if $u_i w_j \in F_1$ this means that $v_i v_j$ are successive in C.
Further, since F is $X-$saturating every point of $V(G)$ has a successive point.
Thus the $1-$factor F_1 of H corresponds to a $2-$factor of G.

2 Decompositions

Definition 5 *Given two graphs G and H, an $\underline{H-\text{decomposition}}$ of G is a collection of graphs, $\{H_1, \ldots, H_k\}$, all isomorphic to H, on the vertex set of G with isolated points removed such that every edge of G appears exactly once in the collection.*

$$\text{i.e. } E(H_i) \cap E(H_j) = \emptyset \ (i \neq j) \ \text{ and } \bigcup_{i=1}^{k} E(H_i) = E(G).$$

Definition 6 (Kirkman 1847, Steiner 1853) *A decomposition of K_n into triangles (C_3) is called a Steiner Triple System STS(n).*

Theorem 7 (Kirkman 1847, Riess 1859) *A Steiner Triple System STS(n) exists if and only if $n \equiv 1, 3 \bmod 6$.*

Definition 8 (Kirkman 1847) *A Factorization of K_n into triples (C_3) is called a Kirkman Triple System, KTS(n).*

Theorem 9 (Ray-Chaudhuri, Wilson 1972) *A Kirkman Triple System KTS(n) exists if and only if $n \equiv 3 \bmod 6$.*

Theorem 10 (Alspach, Schellenberg 1991)) *For a given $k \geq 3$ a factorization of K_n into C_k exists if and only if n is even.*

Note that all the cycles in the decomposition must have the same size.
Öberwolfach problem Given a set of cycle sizes $C_{m_1}, \ldots, C_{m_\ell}$ for a given $n = \sum_{i=1}^{\ell} r_i m_i$ for some integers $r_1, \ldots r_\ell$ find a decomposition of K_n into r_1 cycles of size m_1, ..., m_ℓ cycles of size r_ℓ.

3 Algorithms

3.1 Backtrack

We wish to find a decomposition of a graph $G = (V, E)$ into triples. There are n vertices and m edges.
Since each triple uses 3 edges There will be $m/3$ triples in total.
We denote the set of triples by $\mathcal{B}$.
Assume that we have a total order $\leq$ on V

$$v_1 \leq v_2 \leq \ldots \leq v_n$$

We may use this ordering to induce an ordering on the k-sets of V lexicographically.
Given k-sets $A, B \subseteq V$,

$$A = \{v_{i_1}, v_{i_2}, \ldots, v_{i_n}\},\ B = \{v_{j_1}, v_{j_2}, \ldots, v_{j_n}\}.$$

Where the elements within the sets are ordered from least to greatest. $A \leq B$ if and only if $\exists p \in \mathbb{N}$ such that $\forall q < p, v_{i_q} = v_{j_q}$ and $v_{i_p} \leq v_{j_p}$.

Example 11

1. $\{0,1\} \leq \{0,2\} \leq \{1,1\} \leq \{1,2\}$ etc.
2. $\{0,1,2\} \leq \{0,1,3\} \leq \{0,2,3\} \leq \{0,5,6\} \leq \{1,2,3\}$ etc.

The Algorithm

Input: A graph $G = (V, E)$

Initialization: $\mathcal{B} = \emptyset$

Recursion ($\mathcal{B}$ = triples so far, E = edges remaining)

```
Search(B, E)
    If |B| = m/3 return (success B)
    Find the ≤ first unused pair v_i v_j ∈ E (∉ B ∈ B)
    For each v_k in ≤ order s.t. v_i v_k & v_j v_k ∈ E, (v_i < v_j < v_k)
        If Search(B ∪ {v_i, v_j, v_k}, E − ({v_i v_j} ∪ {v_i v_k} ∪ {v_j v_k}))
            Return (success B ∪ {v_i, v_j, v_k})
    If no such v_k return Fail
```

If we only want existence `Search` exists on success, otherwise this algorithm will enumerate all instances of a triple system.

Pros

- Will find every possible solution exactly once.
- Easily generalizable.
- If there is no solution will report this.

Cons

very . . .

very . . .

very . . .

very . . .

S L O W

The algorithm may range over all $m/3$-sets of 3-sets from V.
There are over 10^{10} STS(19), and those are only the ones that work.

Example 12
Searching for an STS(9):

012								
034								
056								
078								
135								
146	147							
17X	168							
	236			237			238	
	245	247	248	245	246	248	245	246
	27X	258	257	267	258	25X	267	257
		37X	37X	36X	368		36X	367
								458

Solution: 012, 034, 056, 078, 135, 147, 168, 238, 246, 257, 367, 458

3.2 Hill Climb - Random Algorithms

3.2.1 Idea

Hill Climbs are good for solving optimization problems where the minimum *cost* is known.
Given a set Σ of feasible solutions and for each $S \in \Sigma$ we have an associated cost function $c : \Sigma \to \mathbb{R}$.
We would like to minimize the cost c over all instances of $S \in \Sigma$ ie. Find $S \in \Sigma$ such that $c(S) = \min_{R \in \Sigma}(c(R))$
We are also given a set of transformations $T_i : \Sigma \to \Sigma$ which are such that $c(T_i(S)) \leq C(S)$.

Definition 13 *Given* $S \in \Sigma$*:*

- The neighborhood of S $N(S) = \{R \in \Sigma \mid R = T_i(S) \text{ for some } i\}$
- S is a local minimum if $\forall R \in N(S),\ c(S) \leq c(R)$
- S is a global minimum if $\forall\, R \in \Sigma,\ c(S) \leq c(R)$

The key to finding a good Hill climb is to find a good set of non-decreasing transformations T_i. Hill climbs need the random element, if the set of choices is too constrained it will get stuck.

3.2.2 Algorithm

```
Hill()
    Sideways = 0
    Randomly generate S ∈ Σ
    While S is not a global min.  & Sideways < Max Sideways
        Randomly choose R ∈ N(S)
        If c(R) = c(S), Sideways++
        Else Sideways = 0
        S = R
```

```
        If c(S) = min, return S
    End while
    Return Fail
```

Idea is we start with a random $S \in \Sigma$. We randomly move to a neighboring point. The new point will always have a cost less than or equal to the cost of S. Thus cost always either decreases or stays the same.
Problem is that we can get stuck in a local minimum. To avoid this we abandon the search if too many *sideways* moves have been made.
Calling routine then tries a certain number of times before finally giving up.

```
Main()
    While Count < Max Count
        If Hill() Return Success!
```

Thus hill climbs are characterized by two parameters:
`Max sideways` The maximum number of sideways moves before abandoning this search
`Max Count` The maximum number of times Hill is called before giving up.

3.2.3 1-Factors

Input is an even order $k-$regular graph $G = (V, E)$. We will build up partial factors $F_1, F_2, \ldots, F_k$ edge by edge, initially the partial factors are empty $F_i = (V, \emptyset)$. As we add edges to the partial factors we ensure that no edges within a factor meet and that no edge of G is used more than once. A factor F_i is called *live* if it contains unsaturated points (so is not yet a 1-factor), note that if F_i is live it must contain at least 2 unsaturated points. An edge $e \in E$ is *live* if it has not been placed in any of the factors F_i. We say a vertex $x \in V$ is used in a factor F_i if it appears in an edge of F_i.
Σ is the set of possible arrangements of the edges within the partial factors, subject to the conditions that no edges within a factor meet and that no edge of G is used more than once.
c is the number of edges of G placed into partial factors.
We have the following algorithm for moving within Σ, without decreasing c (T).

A_1:

```
Pick a live factor F_i
Pick x ∈ V st x not used in F_i
Pick y ∈ V st y not used in F_i
Add xy to F_i
If xy is already used in another factor, F_j
    Delete xy from F_j
    Return 1
Return 0
```

There is a slight variation one can make:

A_2

```
Pick a factor F_i
Pick x ∈ V st x ∉ F_i
Pick y ∈ V st xy is live (not used)
Add xy to F_i
If y ∈ F_i (so y appears ewith z say)
    Delete yz ∈ F_i.
    Return 1
Return 0
```

It turns out that the best performance is obtained when we randomly choose A_1 or A_2 at each iteration:

```
Count = 0
While Count++ < Max Count
    For i from 1 to k set F_i = ∅
    Sideways = 0
    While there is a live F_i and Sideways++ < Max Sideways
        Randomly choose i = 1, 2
        Do A_i
        If no edge was deleted in A_i Sideways = 0
```

Note that this algorithm may fail even if G has a 1-factorization, but in practice this is extremely rare.

3.2.4 Triple Systems

For triple systems $\Sigma = \{$ any partial design $\} = \{$ any collection of triples such that each pair appears at most once $\}$.
$c(S) = v(v-1)/6 - |\mathcal{B}| = m/3 - |\mathcal{B}|$
Hill climbing for triple systems is very effective, so we can effectively set `Max Sideways` $= \infty$ and `Max Tries` $= 0$. However if we are searching for a more complex object we may require these parameters.
E is the set edges to be covered.
By `Choose` we mean choose a point at random subject to the given conditions.

```
Hill()
    B = ∅
    While |B| < m/3
        Choose x ∈ V with d(x) > 0
        Choose y ∈ V with x y ∈ E
        Choose z ∈ V with x z ∈ E
        If x z ∉ E  (⇒ ∃ a ∈ V, {x, z, a} ∈ B)
            B = B − {x, z, a}
        B = B ∪ {x, y, z}
```

Pros
Very effective for existence.
Very fast. Can find an STS(303) in 1 second on a PC.
Cons
Random algorithm - may fail to find an answer even though there is one.
Cannot do enumeration.
Not easily generalizable - Good T_i are hard to find.
For example there is no known way to generalize this method to find KTS(v) (Factorizations into triples).