Writing: Algorithms and Definitions

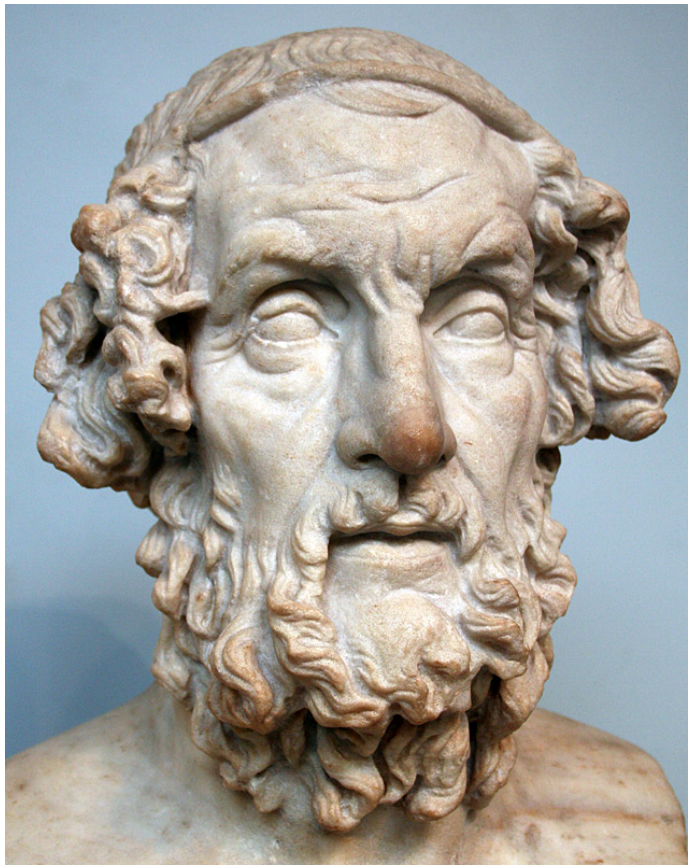
Javier Larrosa

UPC Barcelona Tech

.

On the difficulty of traveling from ideas to text

Consider the following two translations of the first line of the **Odyssey**. It starts describing Ulysses with the Greek word **polytropos**:



Emily Wilson

Tell me about a complicated man...

Robert Fitzerald

Sing in me, Muse, and through me tell the story of that man skilled in all ways of contending...

Algorithms

- In CS we often write algorithms (that we have previously implemented and tested)
- Writing the algorithm differs from implementing it
  - Writing is for **communication purposes** (clarity rather than efficiency)
  - Writing is **done a posteriori** (when key ideas are more clear)
  - Writing means **coding your thoughts** (helps you settle down ideas)

Writing Algorithms: example

Social Networks

Consider a Social Network (e.g. facebook). We are concerned about its users and who they follow.

Celebrities

In a social sub-network a **celebrity** is a person that everybody knows, but does not know anybody.

Goal

Find an efficient algorithm to find **celebrities**

Lets be formal!

Data

Let S be a set of persons and $L(u) \subseteq S$ is the set of people followed by $u \in S$

Celebrities

A user $u \in S$ is a **celebrity** if $|L(u)| = 0$ and for all $v \in S$ such that $v \neq u$, we have that $u \in L(v)$

Goal

Find an **efficient** algorithm to find **celebrities**

Lets be formal!

Abstraction of the problem

Think of the network as a **directed graph** $G = (V, E)$. A **celebrity** is a vertex $c \in V$ such that for every $w \in V$ different from c we have that $(w, c) \in E$ and $(c, w) \notin E$

There are two reasonable data structures for graphs:

1. Matrix of booleans
2. Adjacency lists

In the following we will assume the Matrix of booleans.

Algorithm (ugly)

Assume that users are $1..n$

Function *Celebrity* **is**
    $i := 1$;
    for $j := 2..n$ **do**
        if $j \in L(i)$ **then** $i := j$;
    end
    if $|L(i)| \neq 0$ **then return** false;
    for $j := 1..n$ *s.t.* $j \neq i$ **do**
        if $i \notin L(j)$ **then return** false;
    end
    return true;
end

Algorithm (ugly)

Invariants:

- i may be a celebrity
- $[1..j]$ (except i) are not celebrities

Cost: $O(n)$
if $j \in L(i)$ is $O(1)$ which can be implemented with an array of booleans

Algorithm 2

Two useful properties

1. There is at most one celebrity
2. For any pair of vertices (u, v), if $(u, v) \in E$ then u is not a celebrity, else v is not a celebrity

Algorithm 2

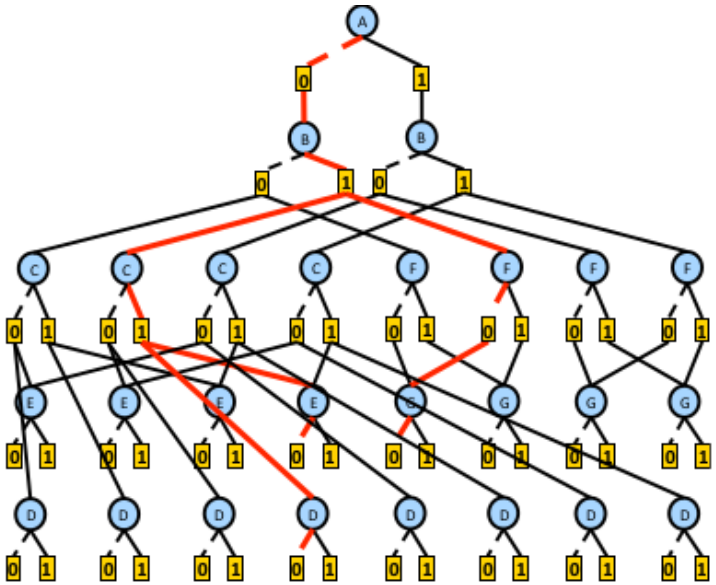
Function *Celebrity* $(G = (V, E))$ **is**
    $S := V$;
    while $|S| > 1$ **do**
        $(i, j) := S.FetchPair()$;
        if $(i, j) \in E$ **then** $S.Push(j)$;
        else $S.Push(i)$;
    end
    return IsCelebrity(G, S);
end
Function *IsCelebrity* (G, v) **is**
    foreach $w \in V$ *s.t.* $w \neq v$ **do**
        if $(w, v) \notin E \vee (v, w) \in E$ **then return** false;
    end
    return true;
end

Invariants: S set of potential celebrities

Writing Algorithms: Example

Search on an AND/OR tree

Find the minimum cost AND/OR solution sub-tree



(b) Example AND/OR search graph. A solution tree is higlighted.

Example

Input: A graphical model $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$, pseudo-tree $\mathcal{T}$, heuristic function $h(\cdot)$
Output: Optimal solution to $\mathcal{M}$

Create root OR node labeled by X_1 and let the stack of created but not expanded nodes $OPEN \leftarrow \{s\}$
Initialize $v(s) \leftarrow \infty$ and the best partial solution tree rooted by s, $T^*(s) \leftarrow \emptyset, UB \leftarrow \infty$
while $OPEN \neq \emptyset$ **do**
    `// Node Expansion`
    Select top node n in OPEN
    if n is an OR node labeled X_i **then**
        foreach $x_i \in \mathbf{D}_i$ **do**
            Add AND child n' labeled $\langle X_i, x_i \rangle$ to the list of successors of n
            Initialize $v(n') \leftarrow 0$ and the best partial solution tree rooted by n', $T^*(n') \leftarrow \emptyset$
    else if n is an AND node labeled $\langle X_i, x_i \rangle$ **then**
        foreach OR ancestor k of n **do**
            Recursively evaluate the cost of the partial solution tree rooted by k, based on the heuristic function $h(\cdot)$, assign its cost to $f(k)$
            `// See Algorithm 2`
            if evaluated partial solution is not better than the current upper bound at k (e.g., $f(k) \geq v(k)$ **then**
                Prune the subtree below the current tip node n
            else
                foreach successor X_j of $X_i \in \mathcal{T}$ **do**
                    Add OR child n' labled X_j to the list of successors of n
                    Initialize $v(n') \leftarrow \infty$ and the best partial solution tree rooted by n', $T^*(n') \leftarrow \emptyset$

Example

```
18  Add successors of n on top of OPEN
    // Bound Propagation
19  while list of successors of node n is not empty do
20      Let p be the parent of n if node n is the root node then
21          return solution: v(n), T*(n)
22      else
23          if p is an AND node then
24              v(p) ← v(p) + v(n), T*(p) ← T*(p) ∪ T*(n)
25          else if p is an OR node then
26              if the new value of v(p) is better than the old one (e.g.
                v(p) > c(p, n) + v(n)) then
27                  v(p) ← c(p, n) + v(n), T*(p) ← T*(p) ∪ ⟨X_i, x_i⟩
28      Remove n from the list of successors of p
29      Move one level up: n ← p
```

Example

Algorithm 2: Recursive Computation of Heuristic Evaluation Function (evalPST)

Input: Partial solution subtree $T(n)$ rooted at node n, heuristic function $h(\cdot)$
Output: Heuristic evaluation function value $f(T(n))$

```
1  if n has no successors then
2      if n is an AND node then
3          return 0
4      else
5          return h(n)
6  else
7      if n is an AND node then
8          Let k_1 ... k_l be the OR children of n
9          return sum_{i=1}^{l} evalPST(T(k_i), h(·))
10     else if n is an OR node then
11         Let k be the AND child of n
12         return c(n, k)+ evalPST(T(k), h(·))
```

Example

```
1  Function BBor(n, ub)
2  begin
3  |  for m ∈ ch(m) do
4  |  |  if c(n, m) + h(m) < ub then
5  |  |  |  ub := c(n, m)+BBand(m, ub − c(n, m))
6  |  |  end
7  |  end
8  |  return ub;
9  end

10 Function BBand(n, ub)
11 begin
12 |  if ch(n) = ∅ then return 0;
13 |  foreach m ∈ ch(n) do q(m) := h(m);
14 |  foreach m ∈ ch(n) do
15 |  |  if Σ_{m'∈ch(n)} q(m') ≥ ub then return ub;
16 |  |  q(m) :=BBor(m, ub − Σ_{m'∈ch(n),m'≠m} q(m));
17 |  end
18 |  return Σ_{m∈ch(n)} q(m);
19 end
```

On Algorithms

How to communicate a non-trivial algorithm

If the algorithm is important, give its pseudo-code as a Figure and describe it in words in the text showing the connection between the two

Pseudo-code

ALGORITHM 1: BT algorithm

```
1  BT ALGORITHM(G, cliqueCost, mergeCost)
2  Π ← EnumeratePMCs(G) [7, 17];
3  T ← {};
4  foreach Ω ∈ Π do
5      foreach D ∈ C(G \ Ω) do
6          S ← N(D);
7          C ← The component of G \ S such that Ω ⊂ S ∪ C;
8          T ← T ∪ {(Ω, S, C)};
9      end
10     T ← T ∪ {(Ω, ∅, V(G))};
11 end
12 sort T in increasing order of |S ∪ C| ;
13 dp[(S, C)] ← ∞ for all (S, C);
14 foreach (Ω, S, C) ∈ T do
15     cost ← cliqueCost(Ω, S);
16     foreach C' ∈ C(G[C \ Ω]) do
17         S' ← N(C');
18         cost ← mergeCost(cost, dp[(S', C')]);
19     end
20     if cost < dp[(S, C)] then
21         dp[(S, C)] ← cost;
22         optChoice[(S, C)] ← Ω;
```

Algor. Description

4.2 Detailed Description

Our implementation of the BT algorithm is presented in pseudocode as Algorithm 1. The implementation is mainly based on [7, 17, 23]. As mentioned, the BT algorithm works by decomposing the computation of $f(G) = f(R(\emptyset, V(G)))$ into the computation of $f(R(S,C))$ of all blocks of G. Furthermore, following Corollary 1, the value of $f(R(S,C))$ is computed as the minimum cost of $R(S,C)$ with respect to Ω over all $\Omega \in \Pi(G)$ satisfying $S \subsetneq \Omega \subset (S,C)$.

Algorithm 1 proceeds over triplets of form (Ω, S, C), where (S,C) is a block and $\Omega \in \Pi(G)$ satisfies $S \subsetneq \Omega \subset (S,C)$. The optimal cost of $R(S,C)$ with respect to Ω is computed on Lines 14–18. Whenever this cost is lower than the best known cost for $R(S,C)$ (Line 20), the value of $dp[(S,C)]$ is updated (Line 21) and Ω is stored in $optChoice[(S,C)]$ (Line 22). After processing all triplets (Lines 16–22), the value of each $dp[(S,C)]$ is equal to $f(R(S,C))$ for all blocks, and $optChoice[(S,C)]$ contains the PMC $\Omega \subset S \cup C$ that needs to be completed into a clique when constructing a optimal triangulation of $R(S,C)$. Specifically, the value of $f(G)$ is stored in $dp[(\emptyset, V(G))]$ (Line 25).

After running Algorithm 1, the optimal triangulation H can be reconstructed using a breadth-first search like procedure shown in Algorithm 2. Starting from $B = (\emptyset, V(G))$ (Line 3), the potential maximal clique Ω_B corresponding to $f(R(B))$ is completed into a clique (Line 7), and all blocks $B_i \in (B : \Omega_B)$ are added to the queue (Lines 8–10). Notice that $\Omega_B = optChoice[B]$.

We note that in order to compute the optimal cost of $R(S,C)$ for a PMC Ω, the optimal cost of all blocks in $(S, C : \Omega)$ needs to be computed. In our implementation of Algorithm 1, all triplets (Ω, S, C) are computed before the actual search (Lines 2–10), and then processed in order of increasing sizes of $S \cup C$ (Line 12). First all potential maximal cliques are enumerated using the procedure from [17] (Line 2) which in turn uses the procedure for enumerating all minimal separators from [7]. Then, for each potential maximal clique, all blocks (S,C) for which $S \subsetneq \Omega \subset (S,C)$ are initialized (Lines 4–10). The addition of the "dummy state" $(\Omega, \emptyset, V(G))$ on Line 10 is used for retrieving the optimal value $f(G)$.

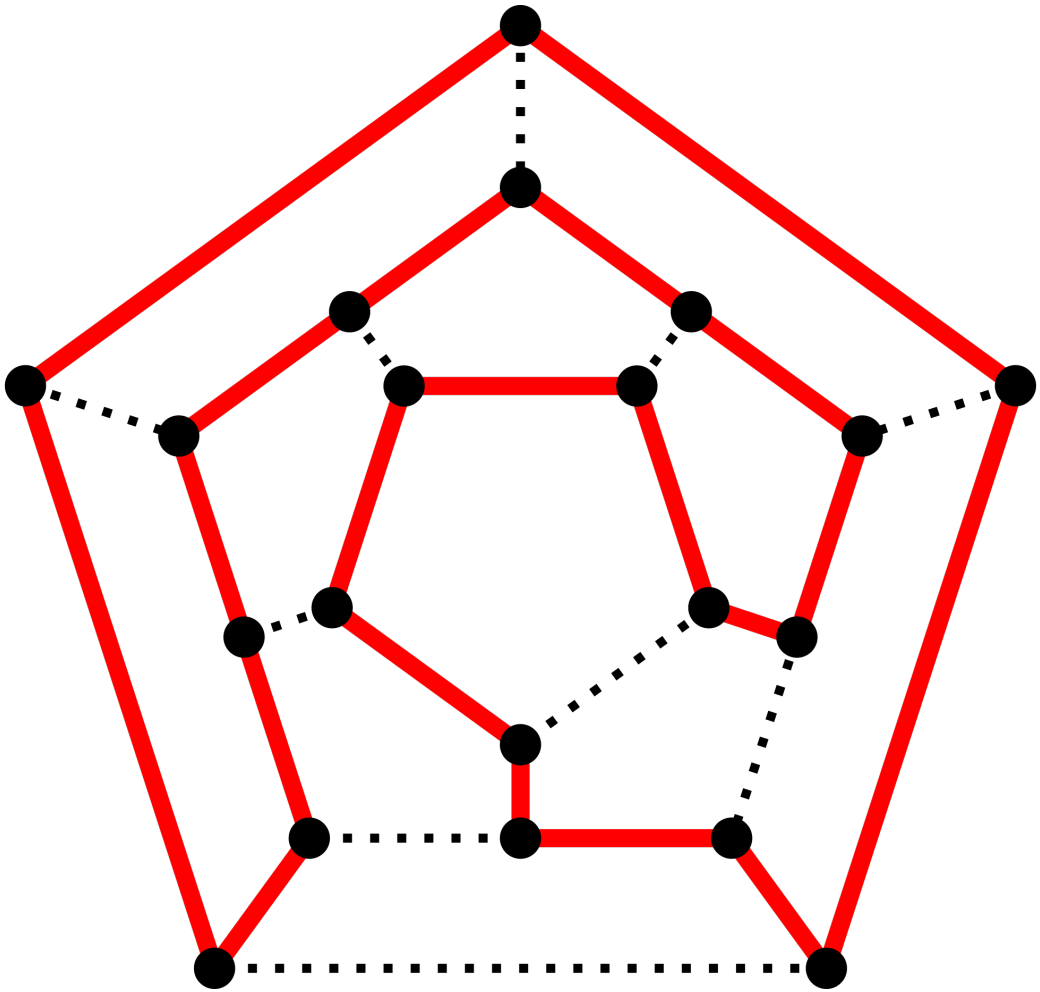
We do not present pseudocode for enumerating potential maximal cliques here, as our implementation is directly based on the pseudocode of [17], using also the optimizations mentioned therein. Let G be a graph with n nodes, $v \in V(G)$ and $G' = G \setminus \{v\}$, i.e., G with the node v removed. The enumeration of potential maximal cliques is based on a characterization of $\Pi(G)$ in terms of $\Pi(G')$, $\Delta(G')$ and $\Delta(G)$. In other words, the set $\Pi(G) = \Pi(G[V_n])$ is computed by iteratively computing

Definitions

In Scientific Writing, we often have to write **definitions**

- Definitions can be given in **words** or in **mathematical writing**
- Sometimes we give both
- Sometimes we add an example

Hamiltonian Graph



Hamiltonian

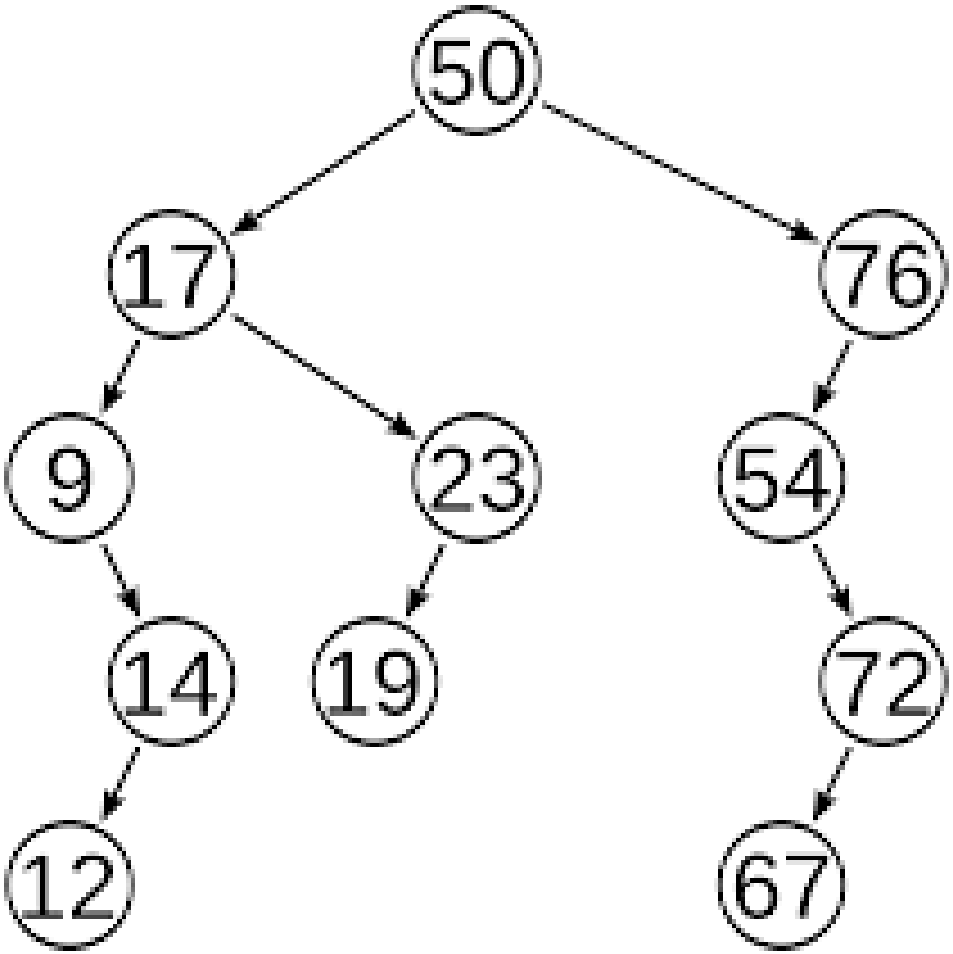
Hamiltonian

A graph is Hamiltonian if it contains a **loop** such that every vertex in the graph is **visited** exactly once

Hamiltonian (more formal)

Let $G = (V, E)$ be an undirected graph with $V = \{1, 2, ..., n\}$. A *path* of lenght k is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that (v_i, v_{i+1}) is in E. A *cycle* is a path that starts and ends at the same vertex (i.e, $v_1 = v_k$). A cycle is *proper* if the only repetition is the starting/ending vertex.
We say that G is *Hamiltonian* if there is a proper cycle of length $n + 1$

Binary Search Tree



Assume knowledge on trees (node, left child, right child,...)

Binary Search Tree

Binary Search Tree

Consider binary trees where each vertex has an associated label. In a *binary search tree* every vertex satisfies that all the labels on its left (respectively, its right) are smaller than (respectively, greater than) or equal to its label.

Binary Search Tree

Notation

Consider a binary tree $T = (V, E)$ with *root* $r \in V$. Let $v \in V$ be an arbitrary vertex. The *sub-tree* on the *left* and *right* of v are noted $left(v)$ and $right(v)$, respectively. Let's assume that each vertex has associated a *numerical label* noted $label(v)$.

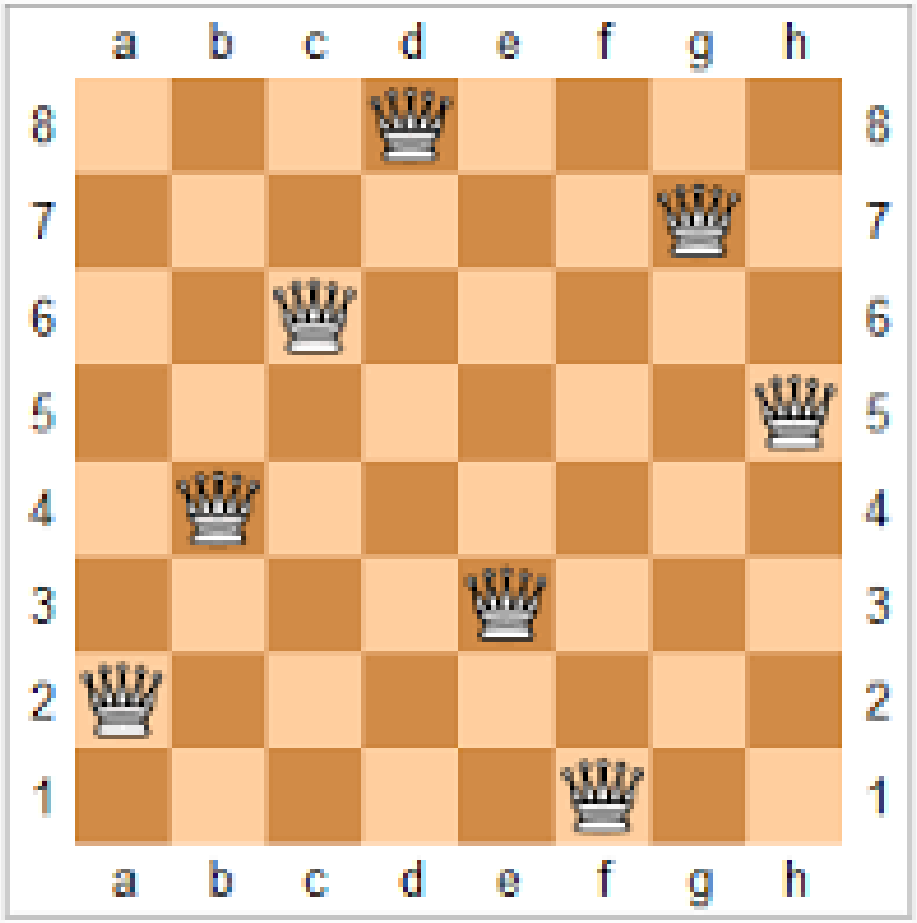
Binary Search Tree (more formally)

We say that T is a *binary search tree* if for every vertex v we have that:

- every label in $left(v)$ is smaller than or equal to $label(v)$
- every label in $right(v)$ is larger than or equal to $label(v)$

Why did I give a name to the root? :-(

n-queens



One solution to the eight queens puzzle

n-queens

n-queens

The n-queens problem consists on placing n **queens** in an $n \times n$ chess board in such a way that no pair of queens **attack** each other. Recall that, according to chess rules, queens can **move** any number of positions along rows, columns and diagonals

n-queens (more formally)

Let $(x_1, x_2, \ldots, x_n)$ be a set of variables taking values in the range $1..n$. The n-queens problem consists on assigning one value to each variable in such a way that for all $i, j = 1..n$, with $i \neq j$:

- $x_i \neq x_j$
- $|x_i - x_j| \neq |i - j|$

n-queens

n-queens (balanced)

The n-queens problem consists on placing n **queens** in an $n \times n$ chess board in such a way that no pair of queens **attack** each other. We represent that with a set of variables $(x_1, x_2, \ldots, x_n)$ taking values in the range $1..n$. Variable x_i taking value j represents a queen placed on cell (i, j). The problem is to assign one value to each variable in such a way that for all $i, j = 1..n$, with $i \neq j$:

- $x_i \neq x_j$ (not in the same column)
- $|x_i - x_j| \neq |i - j|$ (not in the same diagonal)

mastermind

Mastermind

Mastermind is a *code-breaking* game for two players: the *codemaker* and the *codebreaker*. The codemaker makes up a code which keeps secret from the codebreaker. A *code* is a sequence of four colors out of 6 candidate colors. The same color can appear more than once in the code. The goal of the codebreaker is to discover the code as soon as possible during a sequence of guesses. After each guess the codemaker gives some feedback. In particular it tells the codebreaker how many colors have been guessed (irrespective of the position) and how many colors have been guessed in the right position. For example, if the code is (r, b, r, p) and the guess is (g, p, r, g) the feedback will be *you got two right colors, one of them in the right position*. The feedback will be used to refine the codebreaker guess in the subsequent iteration.

Mastermind

Mastermind (more formally)

A *code* is a quadruple $C = (c_1, c_2, c_3, c_4)$ where each element is a number from 1 to 6. The game of *Mastermind* consists of discovering an unknown code C in a minimum number of guesses. After each guess $G = (g_1, g_2, g_3, g_4)$ some feedback is obtained, which will be used to refine subsequent guesses. The feedback is the following:

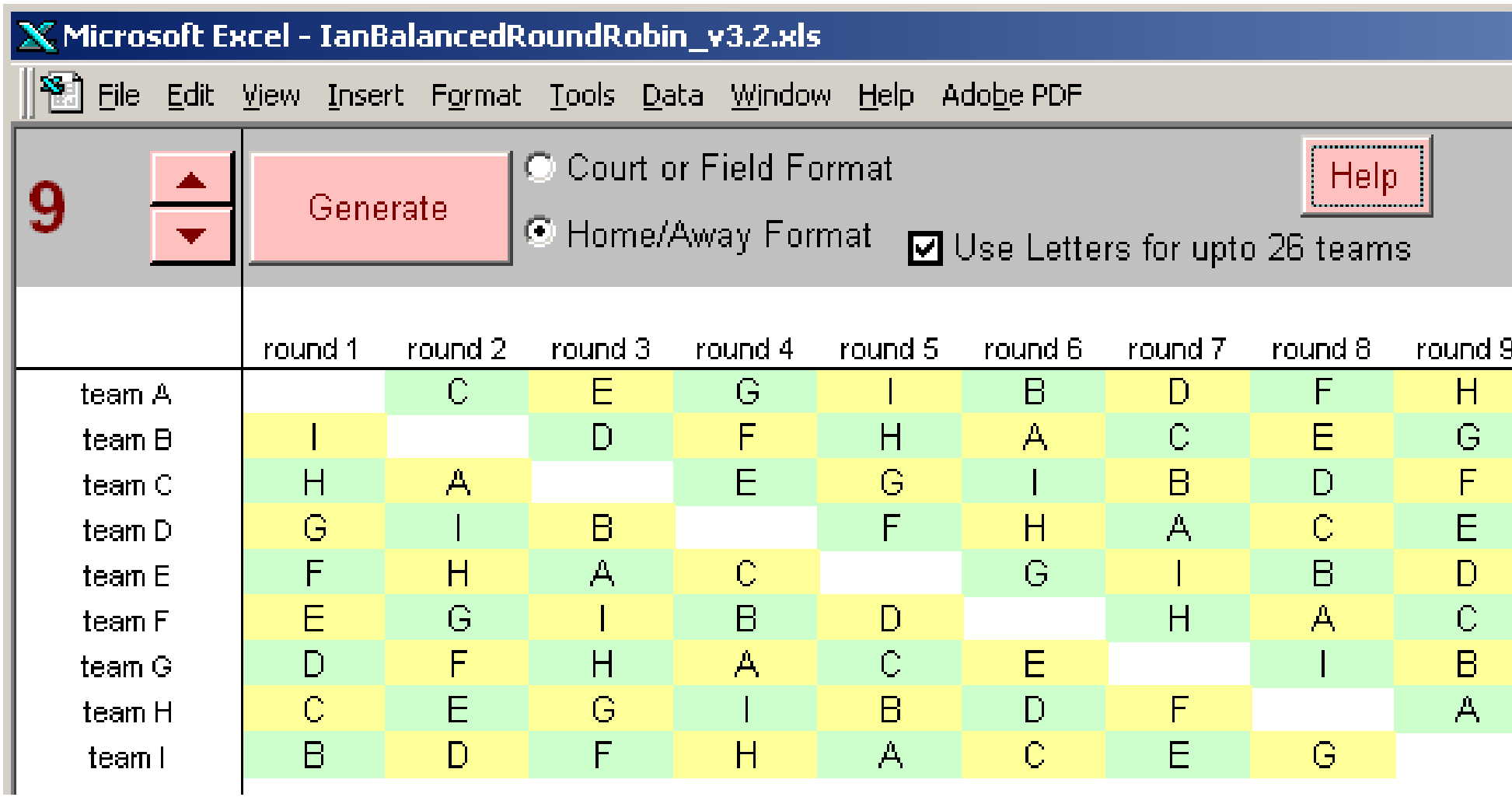
- Number of guessed numbers irrespective of the position. Formally, for each number $i = 1..6$, let C_i be the number of times i occurs in C, and G_i the number of times i occurs in G,

$$\sum_{i=1}^{6} \min\{C_i, G_i\}$$

- Number of positions where the guess matches with the code,

$$|\{1 \leq i \leq 4 |\ g_i = c_i\}|$$

Round Robin Schedule



	round 1	round 2	round 3	round 4	round 5	round 6	round 7	round 8	round 9
team A		C	E	G	I	B	D	F	H
team B	I		D	F	H	A	C	E	G
team C	H	A		E	G	I	B	D	F
team D	G	I	B		F	H	A	C	E
team E	F	H	A	C		G	I	B	D
team F	E	G	I	B	D		H	A	C
team G	D	F	H	A	C	E		I	B
team H	C	E	G	I	B	D	F		A
team I	B	D	F	H	A	C	E	G	

Round Robin Schedule

Round Robin

Consider a two-teams sport for which a tournament with n teams has to be scheduled. A round robin schedule is the pairing of the teams during $n-1$ weeks in such a way that any pair of teams are paired exactly once. We can assume n being an even number, since otherwise we can add a dummy team. Playing against the dummy team represents a resting week

Round Robin Schedule

Round Robin (formal)

Let n be an even natural number. A round robin schedule is a matrix $M_{n \times n-1}$ such that for all $1 \leq i \leq n, 1 \leq j < n$:

- $M(i,j) \in [1..n]$ and $M(i,j) \neq i$
- if $M(i,j) = k$, then $M(k,j) = i$
- $\forall_{1 \leq j < j' < n}$, $M(i,j) \neq M(i,j')$

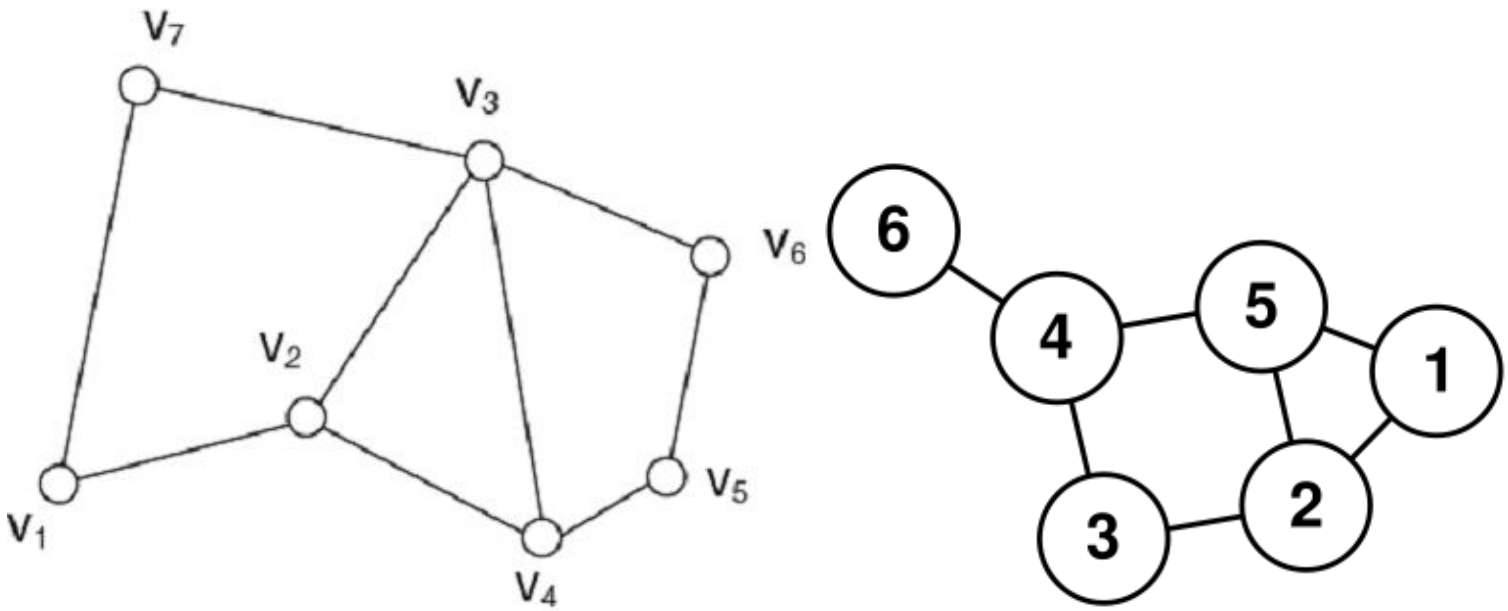
Round Robin Schedule

Round Robin (balanced)

Consider a two-teams sport for which a tournament with n teams has to be scheduled. A round robin schedule is the pairing of the teams during $n-1$ weeks. Let n be an even natural number (if n is odd, we can add a dummy team and playing against the dummy team means not playing that week). The schedule is represented as a matrix $M_{n \times n-1}$ with $M(i,j) \in [1..n]$ denotes who plays against team i at week j. Note that for all $1 \leq i \leq n, 1 \leq j < n$ the following conditions must hold:

- Teams play teams: $M(i,j) \neq i$
- Teams are paired: if $M(i,j) = k$, then $M(k,j) = i$
- Same two teams only play once: $\forall_{1 \leq j < j' < n}$, $M(i,j) \neq M(i,j')$

Induced Width



Induced Width

Induced Width

Let $G = (V, E)$ be an undirected graph with $V = \{1, 2, \ldots, n\}$. The **neighbors** of vertex $v \in V$ is the set of vertices adjacent to v. Its **higher neighbors** are those whose index is larger than v. The **width** of v is the number of higher neighbors that it has. The **width** of G is the maximum width among all its vertices.

The **induced graph** $G^* = (V, E^*)$ is obtained from G as follows: take its vertices in increasing order. When considering vertex v connect with a new edge any pair of higher neighbors that are not already connected.

The **induced width** of G, is the width of its induced graph.

Induced Width

Width

Let $G = (V, E)$ be an undirected graph with $V = \{1, 2, \ldots, n\}$. The **neighbors** of vertex $v \in V$, noted $N(v)$ is the set of vertices adjacent to v,

$$N(v) = \{u \in V |\ (v, u) \in G\}$$

The **higher neighbors** of v, noted $HN(v)$ are those whose index is larger than v,

$$HN(v) = \{u \in V |\ u > v,\ (v, u) \in G\}$$

The **width** of v, noted $w(v)$ is the number of higher neighbors (i.e, $W(v) = |HN(v)|$). The **width** of G is the maximum width among all its vertices,

$$W(G) = \max_{v \in V} W(v)$$

Induced Width

Induced Width

The **induced graph** $G^* = (V, E^*)$ is obtained from G as follows (see Algorithm 3): take its vertices in increasing order. When considering vertex v connect with a new edge any pair of higher neighbors that are not already connected.
The **induced width** of G is the width of its induced graph G^*.

Function *InducedGraph* (G) **is**
    $G^* = (V^*, E^*)$ is a graph;
    $G^* \leftarrow G$;
    foreach $v \in V$ *in increasing order* **do**
        for $u, w \in HN(v)$ **do**
            if $(u, w) \notin E^*$ **then**
                $E^* \leftarrow E^* \cup \{(u, w)\}$;
        end
    end
    return G^*;
end

Algorithm 1: Computation of the induced graph G^* of a graph G. $HN(v)$ denotes the set of neighbors of v with index higher than v in the graph that is being computed G^*

Examples

- https://www.solitairenetwork.com/solitaire/accordion-solitaire-game.html
- https://www.solitairenetwork.com/solitaire/black-hole-solitaire-game.html
- https://dominoes.playdrift.com/
- http://www.playzgame.com/online-flash-games/Eternity-II.php

MineSweeper: 1st ex of bad writing

2 Formal:

Game is based on a matrix M that contains number of cells C. Each cell C can have two sets, CE cell empty or CM cell with mine. Player one sets the matrix that CE+CM=M. Player two selects the cell C that transforms to CE or CM. If it is CM ten the game is over. If it is CE game continues, Some CE have a hint on them that belongs to set 1,2,3. Hint corresponds to number of CM in 8 fields around. Strategy for the first player can be to set matrix M that CM > CE. Tactics for the second player can be to revel hints with a higher number from given set.

MineSweeper: 2nd ex of bad writing

Minesweeper is a two players game in which one player has the role of a **mine layer** and t other of a **mine sweeper**. The game board has a $n \times n$ size, dividing it in n^2 **cells**. The set of ce is C and a cell is represented by c_{ij}, where i is the row and j the column, it has a state $StateC$ and a content $ContentC_{ij}$, where:

- $StateC_{ij}$ = **discovered** *or* **unknow** *or* **flagged**
- $ContentC_{ij}$ = **number** *or* **mine**; number $\in [0..8]$

The number indicates how many mined cells are around, in case the cell doesn't have a mine:

$number \leftarrow 0$;
for $(x \leftarrow i-1;\ x \leq i+1;\ x++;)$ **do**
  for $(y \leftarrow j-1;\ y \leq j+1;\ y++;)$ **do**
    if $ContentC_{xy} = mined$ **then**
      $number++$;
    end
  end
end

MineSweeper: 2nd ex

The mine sweeper can only see $ContentC_{ij}$ **if** $StateC_{ij} =$ discovered, when the game starts the mine layer hides an x number of mines, and sets all the cells to unknown: $\forall c \in C, State_c = unknown$. The mine sweeper has to guess where the mines are, and set $StateC_{ij} =$ discovered . I reveals one mine **he loses immediately**. Mine swapper can also **flag** as many cells as x, numbe of mines, if he thinks there is a mine. **Mine sweepper wins if**: $\forall c \in C, State_c =$ discovered o flagged.

MineSweeper: 3rd ex of bad writing

Minesweeper is a game of two players (p_1, p_2). Given a matrix of size NxM where $N, M\ \varepsilon\ Z^+$ where each of the cells $c_{i,j}$ can contain:

- Mine m
- Number of mines nearby (n) $n\ \varepsilon\ Z^+$ where $n = \sum_{i,j}^{N,M}(Neight(c_{i,j}))\ \varepsilon\ m$. We define a cell neighbourhood as $Neight(c_{i,j}) = c_{i-1,j-1}, c_{i-1,j}, c_{i-1,j+1}, c_{i+1,j}, c_{i+1,j+1}, c_{i+1,j-1}, c_{i-1,j+1}, c_{i,j+1} \parallel 0 \leq i, j \leq N, M.$

When is p_1 turn a player can:

- Reveal a cell $reveal_{p_1}(c_{i,j})$ where $0 \leq i, j \leq N, M$. This makes propagation when $\sum_{i,j=0}^{N,M} costCell(c_{i,j}) = 0 \implies \text{reveal}(c_{i,j})$.

MineSweeper: 3rd ex

- Flag a cell $flag_{p_1}(c_{i,j})$ where $0 \leq i, j \leq N, M$ and $flag(c_{i,j})\ != 1$
- Unflag a cell $unflag_{p_1}(c_{i,j})$ where $0 \leq i, j \leq N, M$ and $flag(c_{i,j}) = 1$

The game ends when p_1:

- Uses $reveal_{p_1}(c_{i,j})$ and p_2 detects that $c_{i,j}\ \varepsilon\ m$.
- $flaggedCells = \sum_{i,j=0}^{N,M} c_{i,j}\ \varepsilon\ m$ and p_2 verifies $c_{i,j}$ are correct.

Otherwise the game continues.

Note that if $c_{i,j}$ is surrounded by $costCell = 1$ then $c_{i,j}\ \varepsilon\ m$. Similarly per bigger costs.