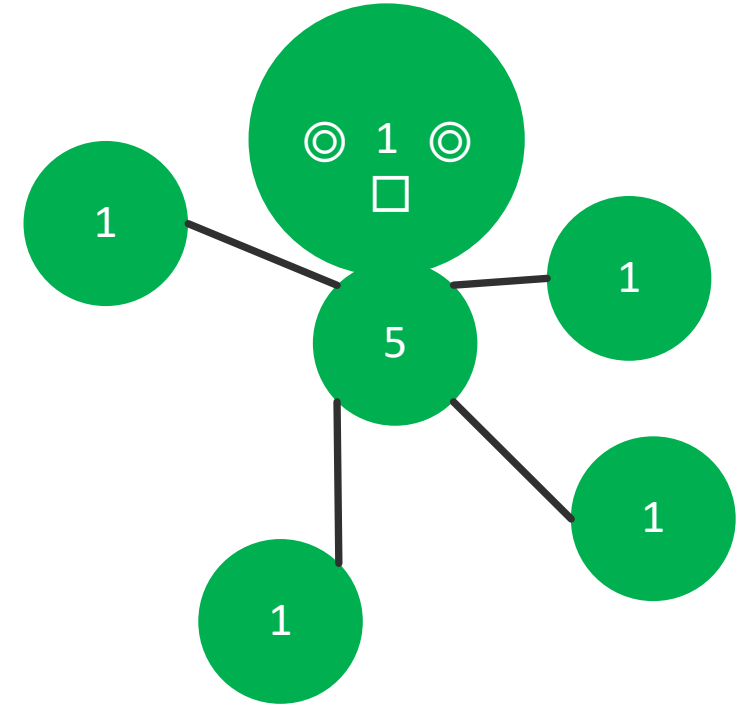


2018/04/06

PGXユーザ勉強会 #7

LT: Green-Marl 入門



クラウドプラットフォームソリューション統括
Cloud Platform ソリューション本部
Big Data & Analytics ソリューション部
中井亮矢

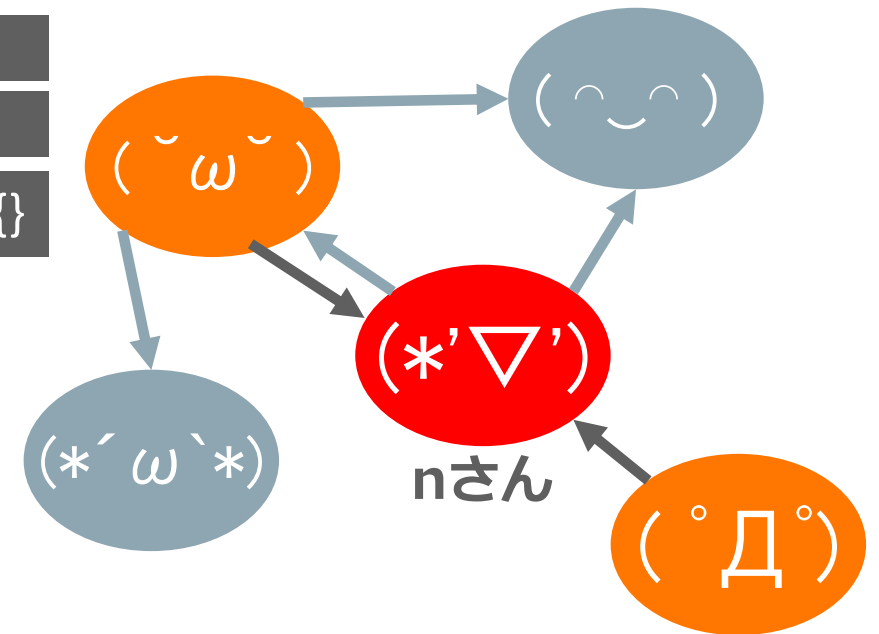
Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

グリーンマる前に

Green-Marl ってなんだ

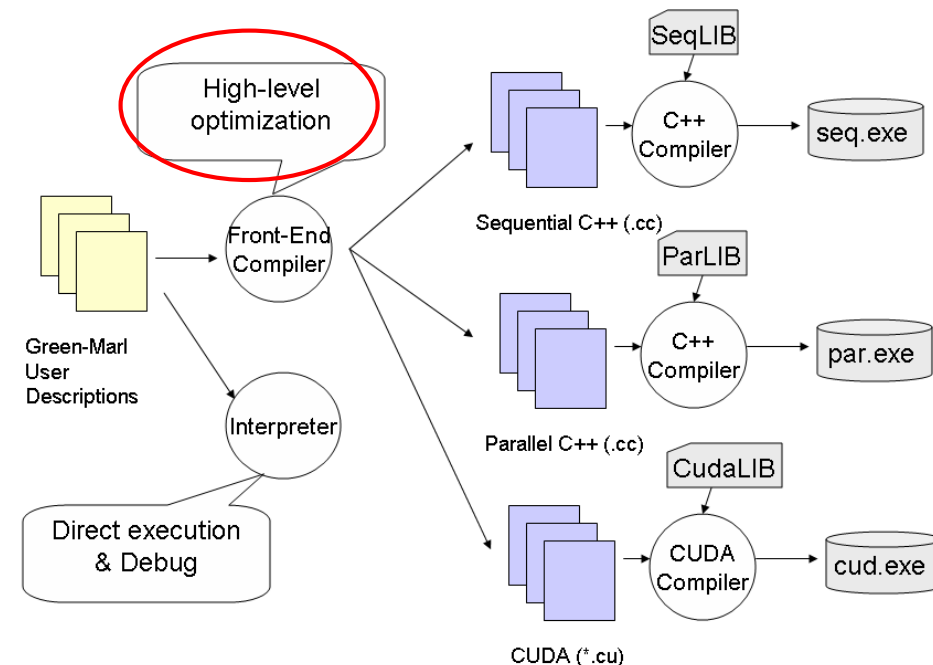
- **グラフ専用のDSL** (Domain Specific Language)
 - グラフを扱う前提の機能特化 (関数、性能)
 - グラフ内のノードリストの取り出す `G.nodes`
 - in関係の隣接ノードのリスト取得 `n.inNbrs`
 - 幅優先探索 `inBFS(x from z){}`
 - グラフ処理向けの最適化
 - 並列探索、
- PGX Shell でコンパイルして実行できる
- 少し癖があるけど**メリット**が大きい



なんで Green-Marl なんだ

- 他のグラフ系 DSLよりイイらしい
- グラフ処理がかなり簡潔に書ける
- グラフ特化DSLならではの最適化
 - 低レベル言語だとできない領域での最適化
 - しょぼい書きっぷりでもある程度空気読んでくれる
- 組み込みアルゴリズムじゃないことがしたい
 - エンティティやエッジの種類が複数あってうまく使って色々したい
 - Built-inアルゴリズムとは違った視点の指標を作りたい
 - 解析作業上、グラフのプロパティを操作して演算とかアレコレしたい

素敵なアルゴリズムは浮かばないけど力業でなんとかしたい



さっそくグリーンマるってみます

どうすりゃいいんだ

- Green-Marlを書く

```
$ vi hello.gm
procedure hello_world() {
    println("Hello World");
}
```

※ 基本の書式

```
procedure プロシージャ名(引数) {
    ...処理
}
```

- compileする

```
$ pgx
hello = session.compileProgram("hello.gm")
```

※ compile時に変数を指定して変数にCompiledProgram (compileProgramメソッドの戻り値)を入れる

※ 例は相対パスですが絶対パス指定もできます

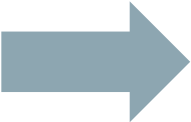
- 実行する

```
:loglevel root INFO
hello.run()
```

※ CompiledProgramのrunメソッドで走る

※ **print文(標準出力)確認のためにのloglevelを上げとく**

※ ログレベルを debugにすると生成されるJavaコードも見れます



```
18:33:49,071 INFO Server - enqueue task
RUN_ANALYSIS
18:33:49,078 INFO App$PrintHelper - Hello World

==> {
  "success" : true,
  "canceled" : false,
  "exception" : null,
  "returnValue" : null,
  "executionTimeMs" : 3
}
```

グラフ的なことをする(1) 最大次数

• グラフ内の最大次数調査

```
procedure max_degree (G: graph): int {  
    return max (n: G.nodes)  
    { n.inDegree() + n.outDegree() };  
}
```

構文のポイント

- 引数は graph
- 戻り値は int (戻り値は : xxx) で宣言の後方に書く
- G.nodes でグラフ内のノードが全部とれる
- ノードの次数は n.degree() で取れる
- max(n: G.noes){ 演算 } で全ノードの演算結果の最大が取れる
- n.inDegree()関数でINの次数が取れる
- n.outDegree()関数でOUTの次数が取れる

```
HERO = session.readGraphWithProperties("hero.json") // グラフ読み込み  
HERO = HERO.simplify() // 重複エッジ排除とか
```

```
maxdeg = session.compileProgram("max_degree.gm", true) // compile  
maxdeg.run(HERO) // 実行  
==> {  
    "success" : true,  
    "canceled" : false,  
    "exception" : null,  
    "returnValue" : 2852,  
    "executionTimeMs" : 3  
}
```


グラフ的なことをする(2) ノードプロパティの追加

• IN次数の移入 (ノードプロパティの追加)

```
proc indegree_centrality(G: graph; IC: N_P<int>)
{
    foreach (n: G.nodes)
        n.IC = n.inDegree();
}
```

構文のポイント

- procedureは procと略してもいい
- N_P (=nodeProp)はノードのプロパティ
- プロパティは型を指定(例では int)
- foreachでくるくる回る
- プロパティへの代入は n.IC = で簡単

```
indeg = session.compileProgram("indegree_centrality.gm") // compile
np_indeg = HERO.createVertexProperty(PropertyType.INTEGER, "np_indeg") // ノードプロパティの生成

indeg.run(HERO, np_indeg) // 実行
HERO.queryPsql("SELECT min(n.np_indeg), max(n.np_indeg) WHERE (n)").print() //確認
```

min(n.np_indeg)	max(n.np_indeg)
0	1427

※Green-Marl内でプロパティを追加する場合には、あらかじめ追加して引数で渡す

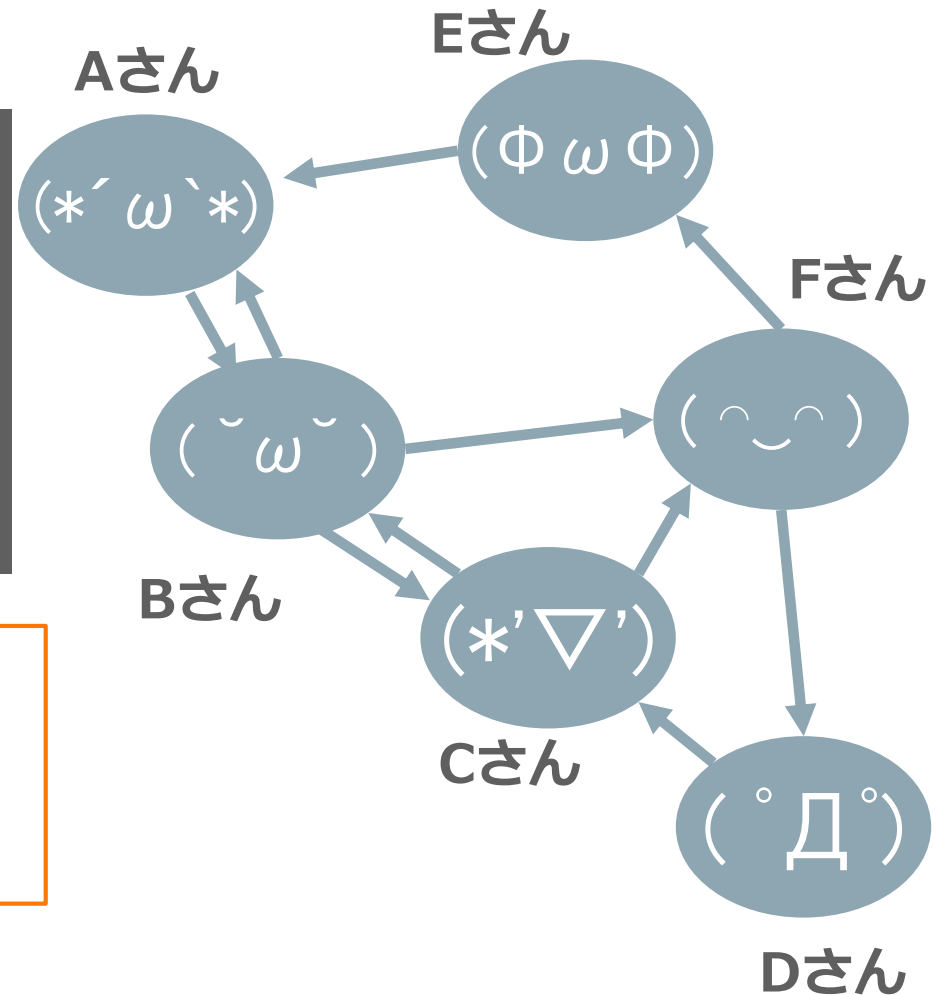
グラフ的なことをする(3) 探索

- 幅優先探索(BFS)と深さ優先探索(DFS)

```
proc trav(G: graph, s_n :node){  
  inDFS(n: G.nodes from s_n){ // 深さ優先探索  
    println("inDFS:" + n );  
  }  
  inBFS(n: G.nodes from s_n){ // 幅優先探索  
    println("inBFS(" + currentBFSLevel() + "):" + n );  
  }  
}
```

構文のポイント

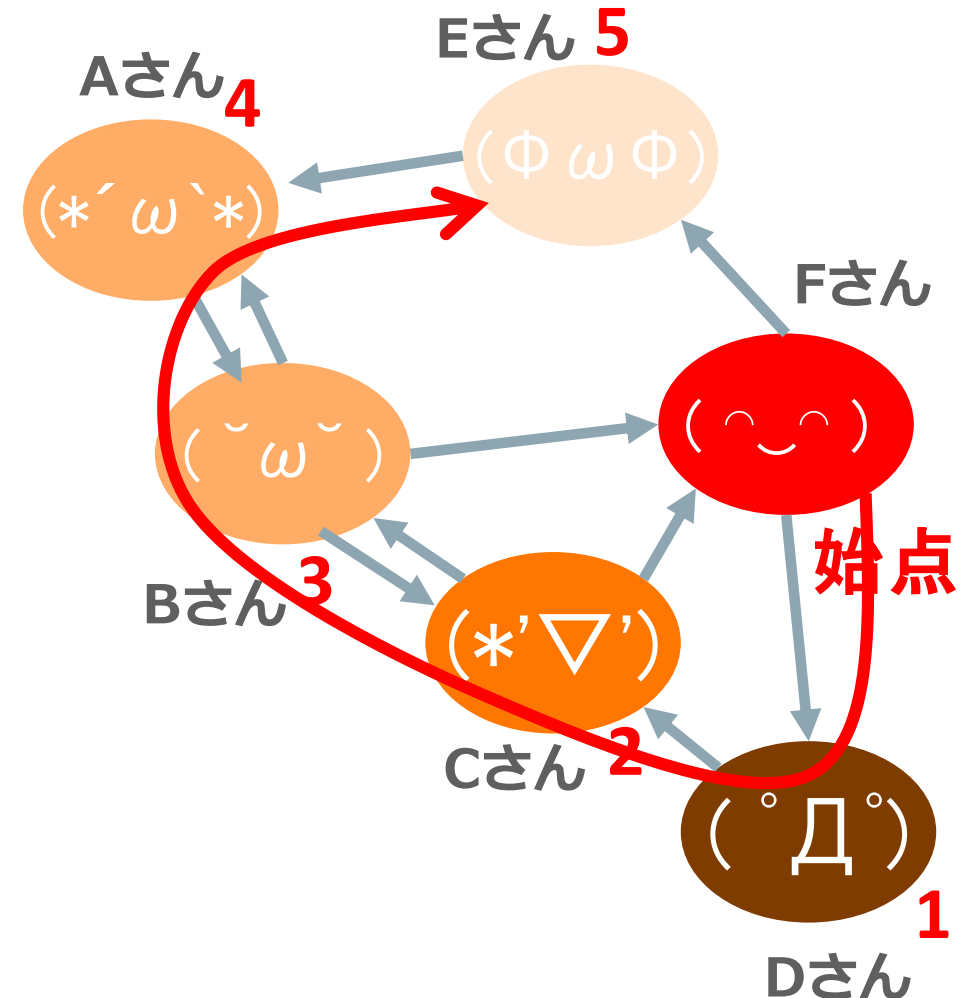
- 探索順にノードをゲットできる
- 深さ優先探索はinDFS
- 幅優先探索はinBFS
- 幅優先探索中は currentBFSLevel() で現在の深さが取れる



グラフ的なことをする(3) 探索 - 深さ優先探索の結果

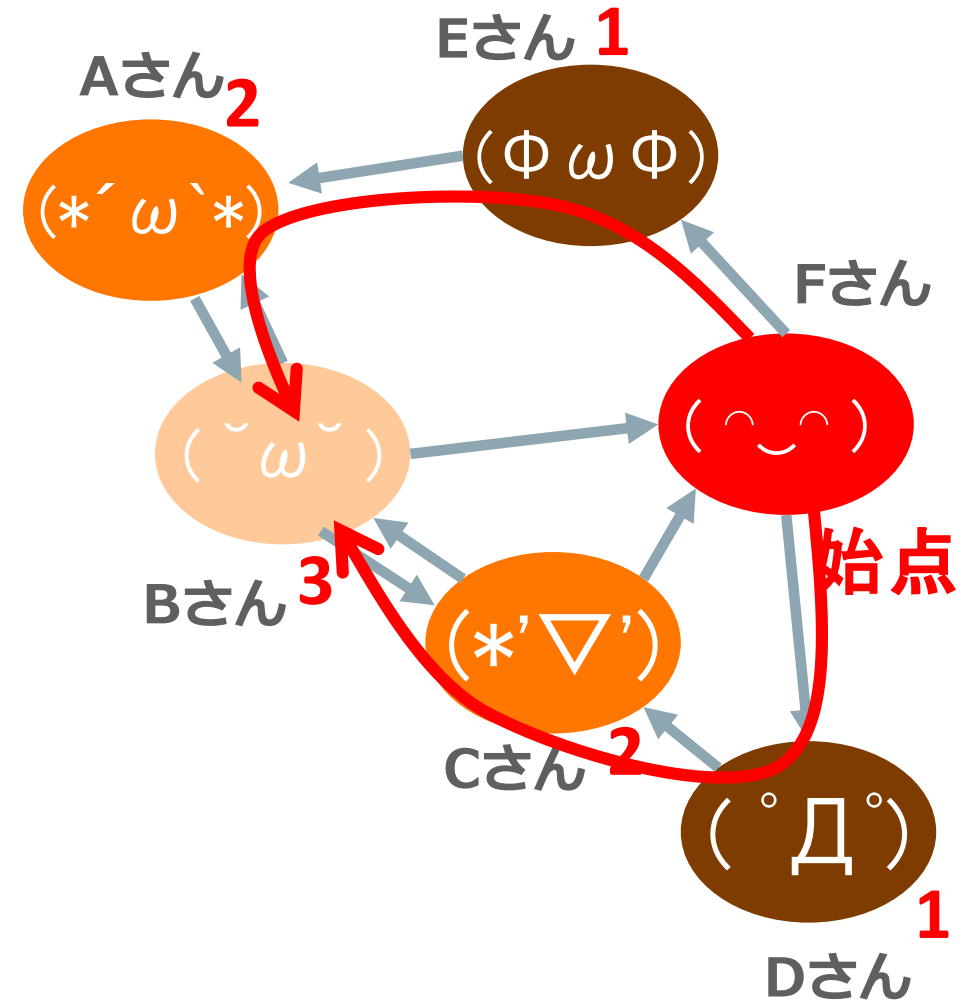
```
pgx> :loglevel root info
pgx> nodeF = G.getVertex("F")
pgx> trav.run(G,nodeF)
```

```
20:32:36,223 INFO App$PrintHelper - inDFS:F
20:32:36,223 INFO App$PrintHelper - inDFS:D
20:32:36,223 INFO App$PrintHelper - inDFS:C
20:32:36,223 INFO App$PrintHelper - inDFS:B
20:32:36,223 INFO App$PrintHelper - inDFS:A
20:32:36,224 INFO App$PrintHelper - inDFS:E
```



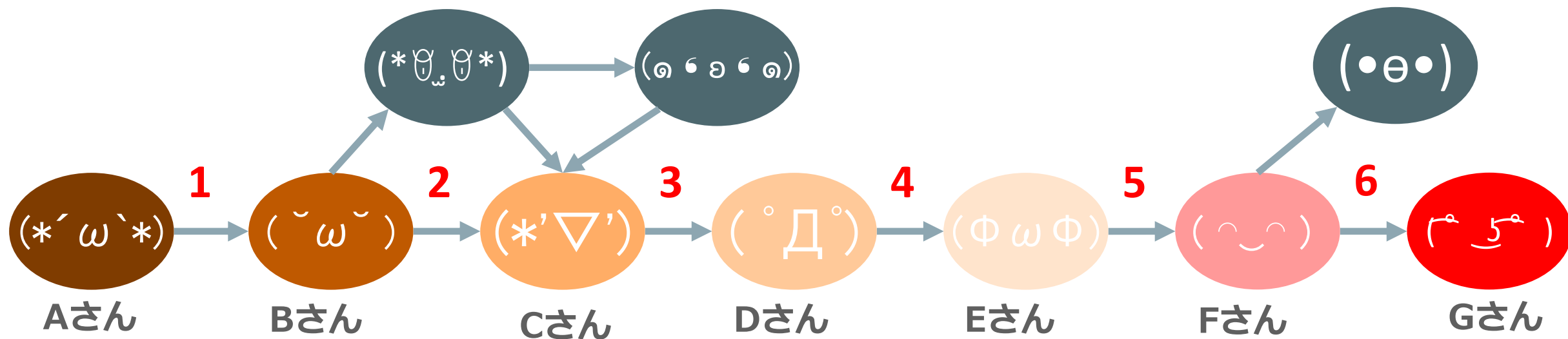
グラフ的なことをする(3) 探索 - 深さ優先探索の結果

```
20:32:36,222 INFO App$PrintHelper - inBFS(0):F
20:32:36,222 INFO App$PrintHelper - inBFS(1):D
20:32:36,222 INFO App$PrintHelper - inBFS(1):E
20:32:36,222 INFO App$PrintHelper - inBFS(2):C
20:32:36,222 INFO App$PrintHelper - inBFS(2):A
20:32:36,222 INFO App$PrintHelper - inBFS(3):B
```



六次の隔たり

六次の隔たり（ろくじのへだたり、Six Degrees of Separation）とは、全ての人や物事は6ステップ以内で繋がっていて、友達の友達…を介して世界中の人々と間接的な知り合いになることができる、という仮説。多くの人からなる世界が比較的少ない人数を介して繋がる[スモール・ワールド現象](#)の一例とされる。[SNS](#)に代表されるいくつかのネットワークサービスはこの仮説が下地になっている。



六次の隔たり調査

全ノードに対して、六次の隔たりに収まっているかをチェックしてカウント、
ついでに収まってない関係を持つノードにフラグを立ててみる。<-後で調査用

```
proc sixdeg(G: graph; out_of_six: N_P<int> ): int {  
    int cnt=0;  
    G.out_of_six = 0;  
    for (s: G.nodes) {  
        inBFS(n: G.nodes from s) (n != s and currentBFSLevel() > 6) {  
            cnt++;  
            s.out_of_six = 1;  
            n.out_of_six = 1;  
        }  
    }  
    return cnt;  
}
```

構文のポイント

- G.nodeProperty = xx で全部初期化
- 全ノードの組み合わせで幅優先探索
- フィルター条件は()内に記述
- 6次以上が発生したら一応カウント
- cntが0なら六次の隔たりが証明

ヒーローネットワークの六次の隔たり

- 六次の隔たり 破れたり、、、

```
pgx> out_of_six = HERO.createVertexProperty(PropertyType.INTEGER, "out_of_six")
pgx> sixdeg.run(HERO, out_of_six)
==> {
  "success" : true,
  "canceled" : false,
  "exception" : null,
  "returnValue" : 418,
  "executionTimeMs" : 5914
}
```

ヒーローネットワークの六次の隔たり

- undirect忘れてました

```
pgx> HERO=HERO.undirect()  
pgx> sixdeg.run(HERO,out_of_six)  
==> {  
  "success" : true,  
  "canceled" : false,  
  "exception" : null,  
  "returnValue" : 0,  
  "executionTimeMs" : 10244  
}
```

- 仮説通りのネットワークでした

PGX Shell(Groovy)との使い分け

- **PGX Shell(Groovy)**

- Green-Marlに欠けてる**汎用言語的な処理**はこちら
 - 型変換とかCalendar処理とか諸々
- **PGQL**結果のリストを使った再演算などを書く場合
- ただし処理性能は自分で考えないとダメ、コードも少し長くなりがち

- **Green-Marl**

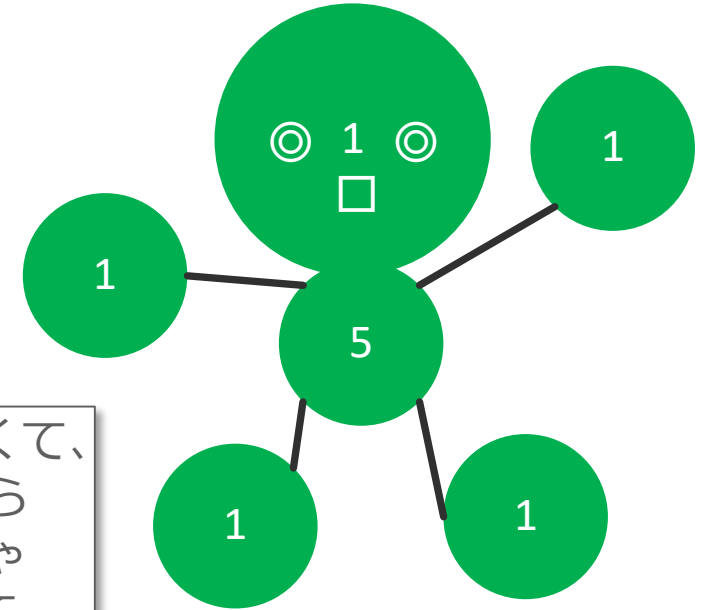
- **探索しながらの演算や処理**に向いてる感じ
- 記述はかなり簡素
- グラフ探索、構造解析、プロパティを使った演算処理等を書くならこちら
 - 自動的に最適化されたコードを生成してくれるからへっぽこな書き方でも助けてくれる

※どっちでもいいケースも結構あるかも

まとめ

- **グラフDSL Green-Marl** の入門編でした
- 基本的な構文と使い方をお伝えしました

「このグラフのアレはどうなってるかな、、、PGQLだとAND条件じゃなくて、ここは出力しなくていいし、このプロパティがない時はあのプロパティから差分で、、、、なんかややこしい、、、、データはあるのに、、、、まあしゃあないか、Groovyで書くか、、えー、プロパティゲットして、forで回して、、あーなんか、長くてめんどい、うーん、、あーもう、」



って時にはすごいいい感じです。

- グラフが思い通りに解析できない時は、是非、**グリーンマ**るってみてください
 - そのうち簡単リファレンスとかチートシート作ってもいいかも

ORACLE®