Decomposition of Cartesian Graph Products

Marc Hellmuth* and Marek Staude**

*,** Department of Computer Science, Bioinformatics, University of Leipzig, Haertelstrasse 16-18, D-04107 Leipzig, Germany

mailto: * marc@bioinf.uni-leipzig.de

** marek@bioinf.uni-leipzig.de

Abstract

The *Cartesian* product G□H of graphs G and H is the graph with vertex set V(G)×V(H) and (a,x)(b,y) is edge in E(G□H) whenever either (ab) in E(G) and x=y, or a=b and (xy) in E(H). Every connected graph has a unique prime factor decomposition with respect to the Cartesian product; see [3, Theorem 4.9]. G is called prime if its unique prime factor decomposition has only one factor, that is, G itself.

The implemented algorithm provides the decomposition of cartesian graph products based on the decomposition with respect to the Djokowic-Winkler relation [1] [4] and the tau relation [5]. The program is written in C++ and we used the well-known BOOST graph library. The Algorithm runs in O(mn) time using O(m) space, here m and n denotes the cardinality of the edge set and the vertex set, respectively [3]. The following documentation shows how to use this program and how it is organized.

Table of content

1 How to Use

This program is a terminal based application. After the successful compilation you have different possibilities to start the program. Furthermore it’s possible to enter a graph or cartesian product manual or via source files. Notice that you have to install the BOOST graph library[1] before compiling.

1.1 Start

Run this program with: “<name.exe>”.

Command 1: example for program-start

```
$ cartesianProduct
```

Or commit a directory path: ”<name.exe> < directory path >”. This directory contains several files with factors (graphs).

Command 2: example for program-start with parameter value

```
$ cartesianProduct /homes/bioinf/TestCenter/graphen
```

1.2 Source files

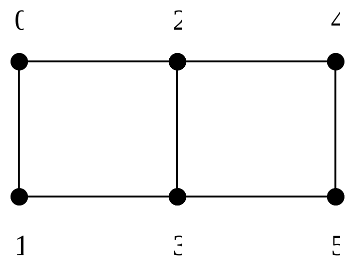
The source file contains vertices and edges of a single graph or cartesian product.
You can choose any file name and file extension but the file content (the exact definition of vertices and edges) is very important.
The following rules should be observed:

1. in case of ‘n’ vertices the vertex number goes from ‘0’ to ‘n - 1’ (without gaps)
2. edges like ‘a b’ and ‘b a’ (a, b are vertices) in one graph are invalid (since we have undirected graphs)

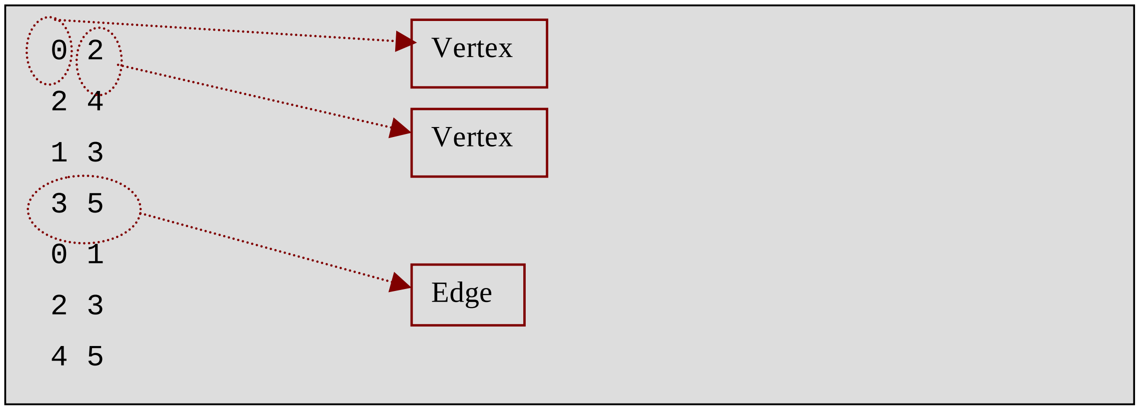
[1] www.boost.org/libs/graph (01/10/07)

The following source file example shows the correct definition of vertices and edges of graph G1:

G1:



Screenshot 1: example of the source file ‘singlegraph.dat’



```
0 2
2 4
1 3
3 5
0 1
2 3
4 5
```

1.3 Menu

After the application started the first screen will show the program menu. In several cases you can choose different ways to enter a graph or cartesian product and end the program.

Screenshot 2: start menu

```
>>>> graph products <<<<

>> press "1" to read a directory with factors,
>>       "2" to read a product,
>>       "3" for manual input or
>>       "0" for program end.
```

1. **Read a directory with factors:**

In the first option you have to specify a *directory* respectively a *directory path* shown in Screenshot 3.

Screenshot 3: read a directory with factors

```
>>>> graph products <<<<

>> press "1" to read a directory with factors,
>>       "2" to read a product,
>>       "3" for manual input or
>>       "0" for program end.
>> 1

>>>> read directory <<<<
>> directory-path: /homes/bioinf/TestCenter/graphen
```

If you have started the application with parameter value the program will select the source directory automatically shown in Screenshot 4.

Screenshot 4: read a directory with factors (parameter value)

```
>>>> graph products <<<<

>> press "1" to read a directory with factors,
>>       "2" to read a product,
>>       "3" for manual input or
>>       "0" for program end.
>> 1

>>>> read directory <<<<
>> directory-path: /homes/bioinf/TestCenter/graphen
>> press ENTER to continue <<
```

2. Read a product:

In the second option you have to specify a *file* respectively a *file path* shown in Screenshot 5.

Screenshot 5: read a file with single product

```
>>>> graph products <<<<

>> press "1" to read a directory with factors,
>>       "2" to read a product,
>>       "3" for manual input or
>>       "0" for program end.
>> 2

>>>> read single product <<<<
>> file-path: /homes/bioinf/TestCenter/singlegraph.dat
```

3. Manual input:

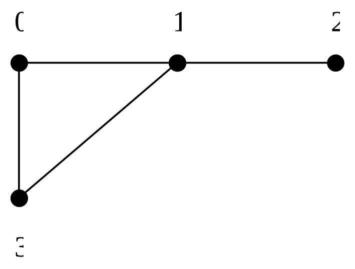
Use option 3 for the manual input of graphs or cartesian products.

In the first step you have to specify the number of edges n. In the second step, you specify source-vertex and target-vertex of n edges (Screenshot 6).

Screenshot 6: manual input of a single product

```
>>>> graph products <<<<

>> press "1" to read a directory with factors,
>>       "2" to read a product,
>>       "3" for manual input or
>>       "0" for program end.
>> 3

>>>> manual input of a product <<<<
>> number of edges: 4

>> edge 0/3 <<
>> vertex source: 0
>> vertex target: 1

>> edge 1/3 <<
>> vertex source: 1
>> vertex target: 2

>> edge 2/3 <<
>> vertex source: 0
>> vertex target: 3

>> edge 3/3 <<
>> vertex source: 3
>> vertex target: 1
```

Result:



The program can take a lot of time for the calculations dependents from the number of vertices. Please be patient :-)

Most of time will be lost in the calculation of the shortest path between all vertices pairs (BOOST-Graph-Library function: floyd_warshall_all_pairs_shortest_paths()).

2 Documentation

2.1 readDirectory()

The function *readDirectory()* read the content (files) of a single directory. The file paths are stored in a vector.

Function 1: readDirectory()

```
vector< string > readDirectory(const char* dir)
```

Input:	const char* dir	// directory path
Output:	vector< string > temp	// vector of files-paths

2.2 ReadAdjacencyList()

The function *AdjacencyList()* read the content of the committed file and generates a graph.

Function 2: ReadAdjacencyList()

```
myGraph ReadAdjacencyList(string sFileName,
                          int flag)
```

Input:	string sFileName	// file path
	int flag	// operation flag (if flag == 2 à add edge-id)
Output:	myGraph g	// single graph

2.3 cartesianProduct()

This function calculates the cartesian product of ‘n’ graphs. In the first step ‘graph_0’ and ‘graph_1’ will be committed. Afterwards the parameter value ‘graph1’ will be the result of the last calculation.

Function 3: cartesianProduct()

```
myGraph cartesianProduct(myGraph graph1,
                         myGraph graph2,
                         vector<vector<vector<int>>>*vecKomp)
```

Input:	myGraph graph1	// graph ‘n’ or cartesian product
		// of calculation
	myGraph graph2	// graph ‘n+1’
	vector<vector<vector<int>>>* vecKomp	// stores the edge coordinates
		// of each graph
Output:	myGraph cart_prod_graph	// new single graph; cartesian product of
		// graph1 and graph2

2.4 getVertexKoord()

The function *getVertexKoord()* generate a vector with the coordinates of all vertices of cartesian product.

Function 4: getVertexKoord()

```
vector< vector< int > > getVertexKoord(myGraph cpgraph,
                        vector<vector<vector<int>>> vecKomp)
```

Input:	myGraph cpgraph	// cartesian product of all graphs
	vector<vector<vector<int>>> vecKomp	// vector of edge coordinates
Output:	vector<vector<int>> vertex	// vector of vertex indices after the
		// calculation of cartesian product

2.5 edgesToVertices()

This Function converts the edges to vertices of the cartesian product. Result is a new graph ‘graph2’ with vertices but without edges.

Function 5: getVertexKoord()

```
void edgesToVertices( myGraph cpgraph,
                      myGraph* graph2,
                      vector< myEdge >* edgeVector)
```

Input:	myGraph cpgraph	// cartesian product of all input-graphs
	myGraph* graph2	// empty graph
	vector<myEdge>* edgeVector	// empty vector
Output:	myGraph graph2	// new graph contains 'n' vertices of 'n'
		// edges of cpgraph
	vector<myEdge> edgeVector	// stores edges (myEdge) of cpgraph

2.6 RELATION_djoko_winkler()

The function *RELATION_djoko_winkler()* weights at first all edges with weight ‘1’. The second part determine shortest path between all vertices pairs. The last part processed Djoko-Winkler-Relation and returns an object of type *myGraph*.

Function 6: RELATION_djoko_winkler()

```
myGraph RELATION_djoko_winkler( myGraph g,
                                myGraph graph2)
```

Input:	myGraph g	// cartesian product graph
	myGraph graph2	// graph with 'n' vertices of 'n' edges of cpgraph
Output:	myGraph graph2	// new graph

2.7 RelationTAU()

The function adds edges between two vertices, if the vertices have only one common neighbour.

Function 7: RelationTAU()

```
myGraph RelationTAU(myGraph graph,
                    myGraph graph2)
```

Input:	myGraph graph	// cartesian product graph
	myGraph graph2	// result of function *RELATION_djoko_winkler()*
Output:	myGraph graph2	// new graph

2.8 decomposition()

The function *decomposition()* extract connected components of cartesian product graph based on BOOST-Library algorithm.

Function 8: decomposition()

```
vector<vector<int>> decomposition( myGraph graph,
                                   myGraph graph2,
                                   vector<myEdge> edgeVector)
```

Input:	myGraph graph	// cartesian product graph
	myGraph graph2	// result of function *RelationTAU()*
	vector<myEdge> edgeVector	// contained edges (myEdge) of
		// cpgraph
Output:	vector<vector<int>> compositionen	// vector of connected-component-
		// elements of cartesian product graph

2.9 outputVertexIndices()

This simple function displayed the vertex indices on the screen.

Function 9: outputVertexIndices()

```
void outputVertexIndices(vector< vector< int > > vertex)
```

Input:	vector<vector<int>> vertex	// vector of vertex indices after the
		// calculation of cartesian product

2.10 findUrGraph()

The function *findUrGraph()* try to extract the initial graphs based on the result of the function *decomposition()* (connected-components).

Function 10: findUrGraph()

```
vector<myGraph> findUrGraph(
                     vector<vector<int>> vecVertexKoord,
                     vector<vector<int>> compositionen,
                     vector<myGraph> graphVector)
```

Input:	vector<vector<int>> vecVertexKoord	// vector of vertex indices after the
		// calculation of cartesian product
	vector<vector<int>> compositionen	// vector of connected-component-
		// elements of cartesian product
		// graph
	vector<myGraph> graphVector	// vector of input graphs (factors)
Output:	vector<myGraph> urGraph	// vector of initial graphs

2.11 extractGraph()

This function extract from cartesian product graph single graphs based on the connected components. The result will be used for the test of isomorphism. The operation based on functions of BOOST-Library.

Function 11: extractGraph()

```
myGraph extractGraph( myGraph graph,
                      vector<int> compositionen)
```

Input:	myGraph graph	// cartesian product graph
	vector<int> compositionen	// vector of connected-component-elements of
		// cartesian product graph
Output:	myGraph graph	// single graph for the test of isomorphism

2.12 testOfIsomorphism()

The function *testOfIsomorphism()* test two graphs of isomorphism. The input graphs are the result (graph) from the *extractGraph()*-function and one of the initial graphs (factors).

Function 12: testOfIsomorphism()

```
bool testOfIsomorphism( myGraph extractedTestGraph,
                        myGraph urGraph)
```

Input:	myGraph extractedTestGraph	// connected-component based graph of
		// cartesian product graph
	myGraph urGraph	// input graphs (same like factors)
Output:	bool isomorph	// true or false

2.13 main()

The *main()*-function is the first operation in each application. All other functions will be called direct or indirect from the *main()*.

Function 13: main()

```
int main(int argc, char* argv[])
```

3 How to Interpret the Results

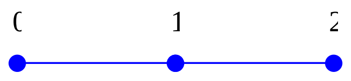
Dependent from the number of input-graphs, the application will display different results.

3.1 Several Graphs

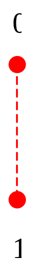
An example of application output after the calculation with several *input-graphs* is shown in the Screenshot 7.

Input-graphs:

G_1:



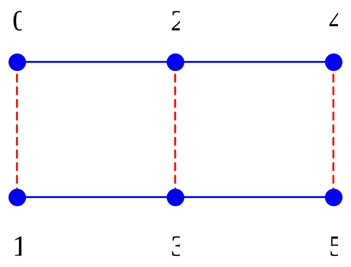
G_2:



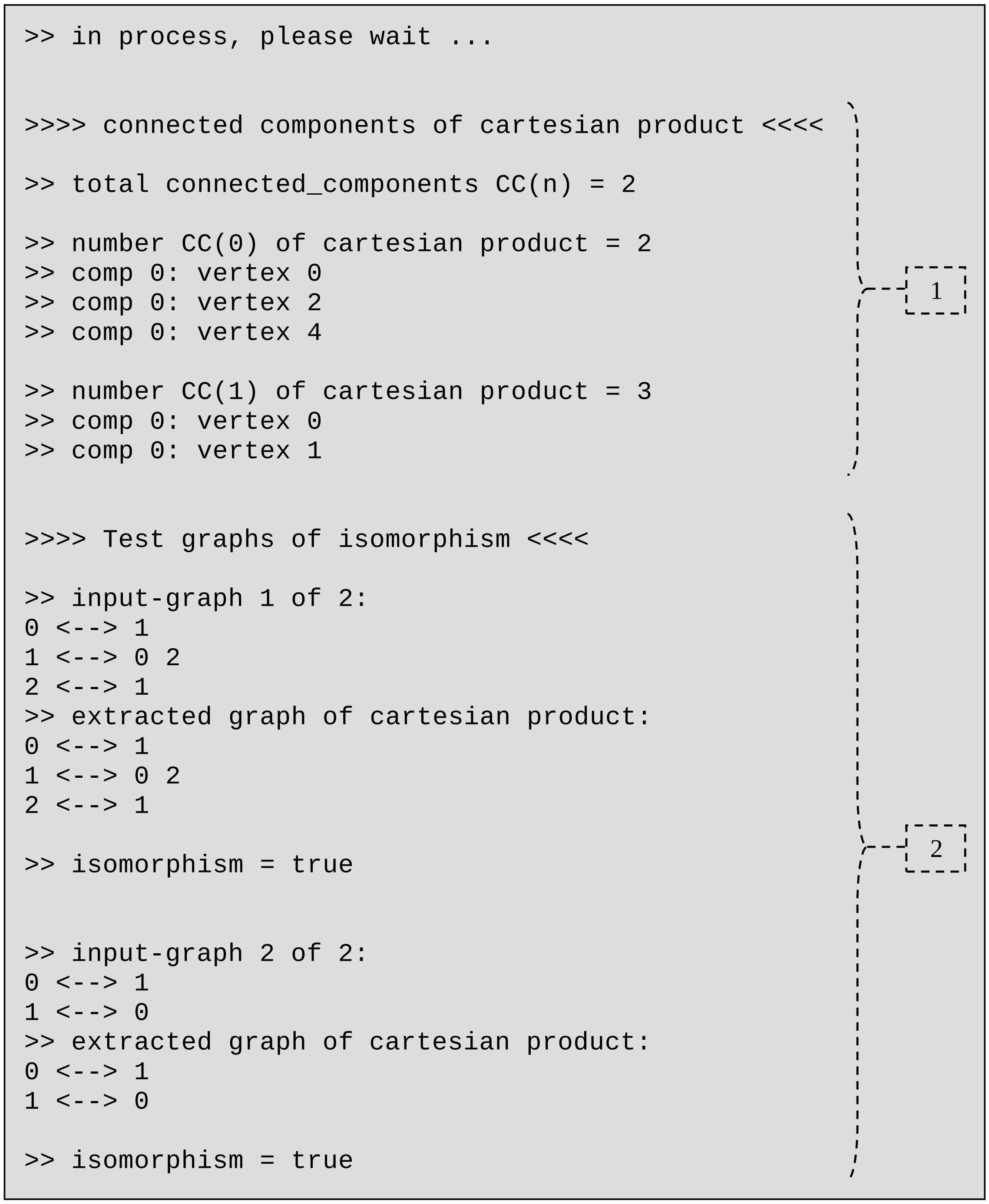
For graphs G_1 and G_2, we define the *cartesian product* $G = G_1 \times G_2$ as follows:

Cartesian product:

G:



Screenshot 7: output after the calculation with several graphs



```
>> in process, please wait ...


>>>> connected components of cartesian product <<<<

>> total connected_components CC(n) = 2

>> number CC(0) of cartesian product = 2
>> comp 0: vertex 0
>> comp 0: vertex 2
>> comp 0: vertex 4

>> number CC(1) of cartesian product = 3
>> comp 0: vertex 0
>> comp 0: vertex 1


>>>> Test graphs of isomorphism <<<<

>> input-graph 1 of 2:
0 <--> 1
1 <--> 0 2
2 <--> 1
>> extracted graph of cartesian product:
0 <--> 1
1 <--> 0 2
2 <--> 1

>> isomorphism = true


>> input-graph 2 of 2:
0 <--> 1
1 <--> 0
>> extracted graph of cartesian product:
0 <--> 1
1 <--> 0

>> isomorphism = true
```

1. Connected components of cartesian product:

The cartesian product be made of n single graphs. After the decomposition we have the connected components of the cartesian product. The connected components can also called “under-graphs” or factors of cartesian product.

total connected_components:

The output *total connected_components* give us the number of factors of the cartesian product.

```
>> total connected_components CC(n) = 2
```

Normally the number is n if n graphs have been committed. In other cases some *factors* (G_1, G_2) of G are also connected graphs itself. The application, specially the function *decomposition()* extract the connected components of the factors, too. So, more than n connected components can be displayed.

number CC(n) of cartesian product:

The first line shows how often the connected component n exists in cartesian product.

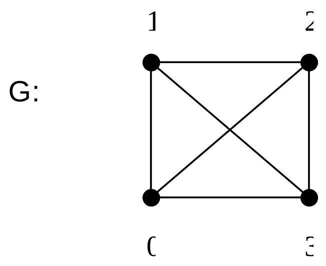
```
>> number CC(0) of cartesian product = 2
```

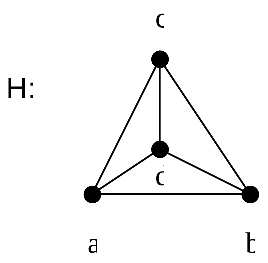
The following lines are the vertices of the first representative (*comp 0*) of the connected component n (CC(n)).

```
>> comp 0: vertex 0
>> comp 0: vertex 2
>> comp 0: vertex 4
```

2. Test graphs of isomorphism:

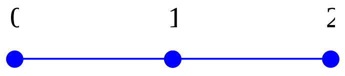
In graph theory, a graph isomorphism is a bijection (a one-to-one and onto mapping) between the vertices of two graphs G and H with the property that any two vertices u and v from G are adjacent if and only if f(u) and f(v) are adjacent in H. If an isomorphism can be constructed between two graphs, then we say those graphs are isomorphic.





The *input-graph* is one of the graphs which were committed at application start.

In this case G_1:



```
>> input-graph 1 of 2:
0 <--> 1
1 <--> 0 2
2 <--> 1
```

The *extracted graph* based on the result of *decomposition()*-function. The connected components are cut out of the cartesien product.

```
>> extracted graph of cartesian product:
0 <--> 1
1 <--> 0 2
2 <--> 1

>> isomorphism = true
```

The isomorphism is true, if the above mentioned definition is applicable.

3.2 Single Graph

Screenshot 8 shows the application output after the calculation with a single graph. The test of isomorphism is not possible, because we have only one graph.

Screenshot 8: output after the calculation with single graph

```
>> in process, please wait ...


>>>> connected components of cartesian product <<<<

>> total connected_components CC(n) = 2

>> number CC(0) of cartesian product = 2
>> comp 0: vertex 0
>> comp 0: vertex 2
>> comp 0: vertex 4

>> number CC(1) of cartesian product = 3
>> comp 0: vertex 0
>> comp 0: vertex 1


>>>> graphs of connected components <<<<

>> graph 1 of 2:
0 <--> 1
1 <--> 0 2
2 <--> 1

>> graph 2 of 2:
0 <--> 1
1 <--> 0
```

For the description of *total connected_components* see “3.1 Several Graphs”.

After the line *graphs of connected components* you can see the connected components of the input graph based on BOOST-Library function *print_graph(graph)*.

Table of Commands

Table of Screenshots

Table of Functions

Bibliography

[1] D. Djoković, Distance preserving subgraphs of hypercubes, J. Combin. Theory Ser. B 14 (1973), pp. 263–267.

[2] W. Imrich and J. Žerovnik, Factoring Cartesian-product graphs, J. Graph Theory 18 (1994), pp. 557–567.

[3] W. Imrich and S. Klavžar, Product Graphs: Structure and Recognition, Wiley, New York (2000).

[4] P. Winkler, Isometric embeddings in products of complete graphs, Discrete Appl. Math. 7 (1984), pp. 221–225.

[5] T. Feder, Product graph representations, J. Graph Theory 16 (1992), pp. 467–488.