

Construction of a Crystal Graph Simulation Engine

by

Shomir Jeffrey Wilson

Honors Thesis Submitted to the Computer Science Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

BACHELOR OF SCIENCE
In Honors Baccalaureate

[PENDING] APPROVED

Dr. Mark Shimosono, Department of Mathematics

Dr. T. M. Murali, Department of Computer Science

This student has fulfilled the requirements of the
Computer Science Department to graduate in Honors Baccalaureate

Dr. Calvin Ribbens, Department of Computer Science

May 2004
Blacksburg, Virginia

Table of Contents

A Brief Introduction to Crystal Graphs	3
Design Overview	6
Class Diagrams	7
User Interface	8
Conclusions	11
Bibliography	12

Appendix

Error Messages	A1
Revision History	A2
Source Code Reference	A3
Vertex Class Hierarchy, Revision 1 (Obsolete)	A10
Vertex Class Hierarchy, Revision 2 (Obsolete)	A11
Interface Commands, Revision 1 (Obsolete)	A12
Interface Commands, Revision 2 (Obsolete)	A13

A Brief Introduction to Crystal Graphs

Crystal graphs are combinatorial representations of objects called *crystal bases*, which are used to solve some problems in statistical mechanics. They are directed graphs with colored edges and uniquely identified vertices. A full examination of the mathematical motivation behind crystal graphs lies beyond the scope of this paper, so we will concentrate on properties relevant to modeling them.

A crystal graph can be partially described as an *l-crystal*, which is a finite directed graph with these properties:

- Each directed edge has a color from the set l . For simplicity, we will refer to an arrow of color $i \in l$ as an *i-arrow*.
- For each vertex v and color i , v has no more than one *i-arrow* entering it. We denote the source of an *i-arrow* into v as the function $e_i(v)$. If no *i-arrow* enters v , then denote $e_i(v) = \emptyset$.
- For each vertex v and color i , v also has no more than one *i-arrow* leaving it. We denote the destination of an *i-arrow* into v as the function $f_i(v)$. If no *i-arrow* leaves v , then denote $f_i(v) = \emptyset$.

These properties allow for components of a graph obtained by eliminating all arrows except for *i-arrows*; these are called *i-strings*. We define two additional functions to examine *i-strings*:

- $\varepsilon_i(v)$ is the number of edges between v and the beginning of its *i-string*.
- $\varphi_i(v)$ is the number of edges between v and the end of its *i-string*.

Functions e , f , ε , and φ allow us to define a *tensor product*. For G_1 and G_2 both *l-crystals*, their tensor product $G_1 \otimes G_2$ is an *l-crystal* as well. Its set of vertices is the Cartesian product of vertices in G_1 and G_2 ; in other words, for every pair of vertices

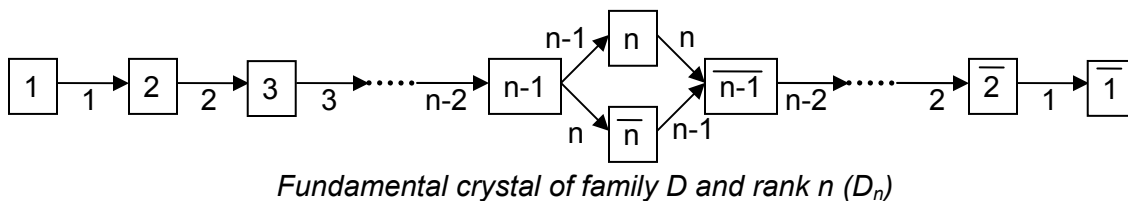
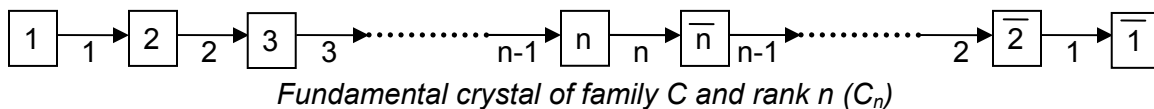
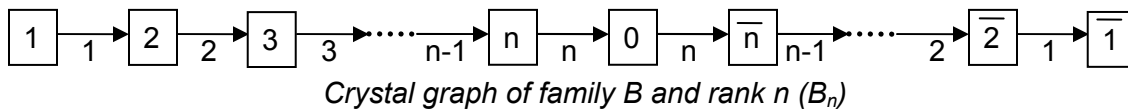
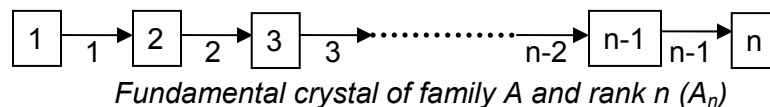
$v_1 \in G_1$ and $v_2 \in G_2$, a vertex identified as $v_1 \otimes v_2$ is contained in $G_1 \otimes G_2$. The arrows between vertices in $G_1 \otimes G_2$ are specified using functions e and f :

$$e_i(v_1 \otimes v_2) = \begin{cases} e_i(v_1) \otimes v_2 & \text{if } \varphi_i(v_1) \geq \varepsilon_i(v_2) \\ v_1 \otimes f_i(v_2) & \text{if } \varphi_i(v_1) < \varepsilon_i(v_2) \end{cases} \quad f_i(v_1 \otimes v_2) = \begin{cases} f_i(v_1) \otimes v_2 & \text{if } \varphi_i(v_1) \geq \varepsilon_i(v_2) \\ v_1 \otimes e_i(v_2) & \text{if } \varphi_i(v_1) < \varepsilon_i(v_2) \end{cases}$$

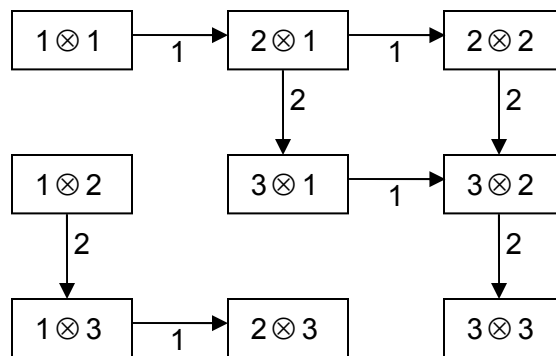
Naturally, $v_1 \otimes \emptyset = \emptyset$ and $\emptyset \otimes v_2 = 0$ for any $v_1 \in G_1$ and $v_2 \in G_2$. The tensor product is associative; that is $(G_1 \otimes G_2) \otimes G_3$ is isomorphic to $G_1 \otimes (G_2 \otimes G_3)$.

An element of a crystal graph is called a *highest-weight vector* if it has no incoming arrows of any color (Kashiwara Bases).

A *root system* describes a set of crystal graphs with similar properties. It consists of a *family* and a *rank*. For the purposes of this project, we will work with four families, known as *A*, *B*, *C*, and *D*. Below are diagrams for the *fundamental crystals* associated with each. The number n in each diagram specifies the arbitrary rank of each crystal graph. Vertices are identified uniquely with numbers 0 or 1 through n , in some cases with bars above them. Numbers below arrows indicate their colors.



The tensor operation can only be applied to graphs of the same root system. Here is an example of the tensor product at work, using the root system A_3 .



Crystal graph $A_3 \otimes A_3$

Note that the result of the tensor product contains i -strings of length greater than one, unlike any of the fundamental crystals on the previous page. For instance, $A_3 \otimes A_3$ contains a 1-string of length two (vertices $1 \otimes 1, 2 \otimes 1, 2 \otimes 2$), as well as two other 1-strings of length one (vertices $3 \otimes 1, 3 \otimes 2$; vertices $1 \otimes 3, 2 \otimes 3$) (Kashiwara “Representations”).

Tensor products with more than two factors (e.g., $A_3 \otimes A_3 \otimes A_3$) are difficult to represent visually. They can be described using the same definitions of e and f as above; however, more efficient algorithms exist to compute their results.

Design Overview

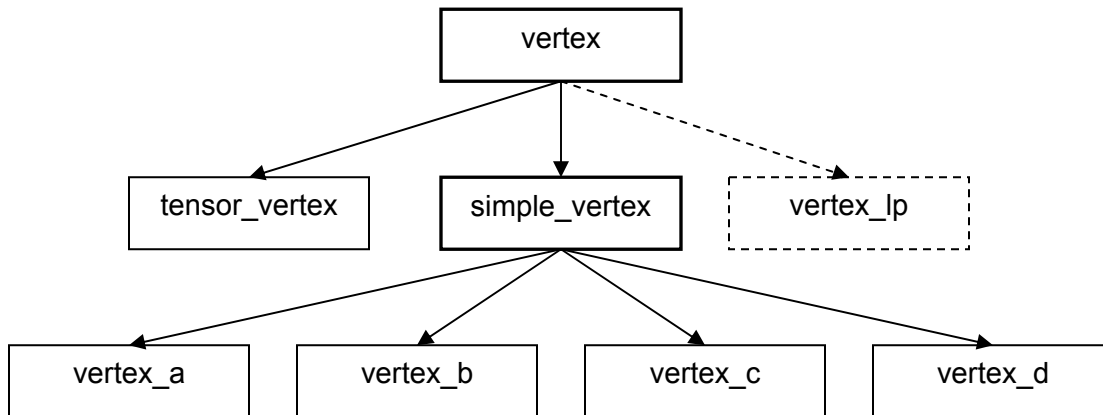
The basic calculations involved in exploring a crystal graph are fairly simple. However, in actual use crystal graphs tend to be large and are constructed out of many tensor products. Manually calculating e or f on a vertex in a graph such as $D_{11} \otimes D_{11} \otimes D_{11} \otimes D_{11} \otimes D_{11}$ is a tedious process at best. This provides motivation for building software to examine crystal graphs.

It was decided early in the design phase that the crystal graph engine should focus on individual vertices rather than entire graphs. Most problems that the engine will be used for are vertex-centric, and the focus on vertices also conserves computing resources. Given only a brief specification of a crystal graph and a valid vertex location within it, any relations that the given vertex has (e.g., i -strings, incoming or outgoing edges, connected vertices) can easily be determined without producing the overall structure of the graph. Accordingly, “navigation” through a crystal graph is accomplished by on-the-fly extrapolation instead of referencing stored structure.

The problem of representing vertices was well-suited for object-oriented design. All crystal vertices have some abstract properties, such as unique location, definitive root system, and relationships with surroundings (via e , f , ε , and φ). Tensor vertices not only share these properties but also are structured recursively in terms of their factors. As defined, a tensor vertex is composed of “smaller” tensor vertices that eventually break down into simple ones. Additionally, other representations of crystal graphs exist, such as Littelmann paths, which use vectors to calculate structure. By utilizing an abstract interface for vertex objects, swapping between representations is easy and almost transparent. Although the Littelmann path method is not yet implemented, adding it will require few modifications to the existing code base.

The command-line user interface was chosen for design simplicity. A graphical user interface—and perhaps graph visualization—will not be difficult to build on top of it.

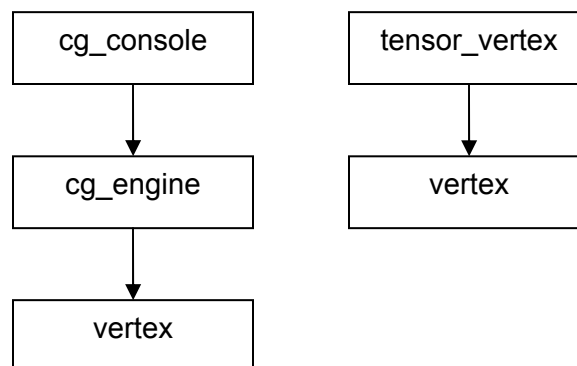
Class Diagrams: Inheritance



Notes:

- Classes *vertex* and *simple_vertex* (bold box) are abstract classes and thus cannot be instantiated.
- Due to time constraints, *vertex_lp* (dashed box) has not fully implemented. This task remains for the next programmer.

Class Diagrams: Aggregation



Notes:

- Summarily, *cg_console* passes syntactically correct commands to *cg_engine*. *cg_engine* manipulates *vertex* objects to produce the desired results, which it passes back to *cg_console* for formatting and display.
- Class *cg_engine* refers almost exclusively to *vertex* for its abstract functionality.
- Class *tensor_vertex* accepts as factors any *vertex* objects, allowing for tensor “trees” of arbitrary depth.

User Interface: Overview

The user interacts with the crystal graph engine via a command-line interface. Typically, the user enters commands and views results directly, but batch processing via output redirection is supported. Input to the interface is case-insensitive; although it is transparent to the user, all input is converted to lower case letters.

For implementation simplicity, this first release of the crystal graph engine supports only single-letter variable identifiers. This provides 26 possible user-defined variables, 'a' through 'z'. An additional identifier, the return register (represented as the dot or '.'), stores the variable created or otherwise referenced in the previous command. The dot can be used wherever the interface expects a vertex identifier.

The list of commands on the following page uses these conventions to express syntax:

- **Bold text** identifies each command name
- i, j , and r represent nonnegative integers
- a, b , and b_n (for any subscript n) represent vertex identifiers
- f represents a specifier for a crystal graph family (letters 'a' through 'd')

As indicated, for some commands the user may enter an asterisk or '*' in place of a vertex identifier. This instructs the engine to apply the command simultaneously to *all* stored vertices.

Following the list of commands is a list of error messages that the engine can display, their internal identifiers (useful chiefly to the programmer), and explanations for each possible error.

User Interface: List of Commands

Command: **big_e** $i a$

Purpose: Recalculates vertex a to the first vertex in its i -string

Command: **big_f** $i a$

Purpose: Recalculates vertex a to the last vertex in its i -string

Command: **copy** $a b$

Purpose: Copies vertex b into vertex a ; replaces a if necessary

Command: **display** a

Result: Displays information on vertex a ; or if a is '*' then information on all stored vertices is displayed

Command: **e** $i a$

Result: Recalculates vertex a to the previous vertex in its i -string

Command: **epsilon** $i a$

Result: Displays the number of steps from vertex a to the beginning of its i -string

Command: **erase** a

Result: Erases vertex a ; or if a is '*' then all declared vertices are erased

Command: **exit**

Result: Ends the session with the engine

Command: **f** $i a$

Result: Recalculates vertex a to the next vertex in its i -string

Command: **make** $a f r n j$

Result: Associates the identifier a with a singleton vertex of family f and rank r at the location specified by "vertex number" r and bar status j (1 if the vertex has a bar, 0 if it does not)

User Interface: List of Commands (*Continued*)

Command: **phi** $i a$

Result: Displays the number of steps from vertex a to the end of its i -string

Command: **tensor** $a f r n b_1 b_2 b_3 \dots b_n$

Result: Associates the identifier a with a tensor vertex of family f and rank r at the location specified by the series of n vertices $b_1 b_2 b_3 \dots b_n$ (vertices should not be separated by spaces in the sequence)

Conclusions

As of the writing of this paper, the latest version of the crystal graph engine is 1.1. It is sufficiently functional for release, although with one caveat.

The core of the engine—*cg_engine* and the *vertex* class hierarchy—is stable and produces correct results. The most gainful achievement of the engine is the ability to calculate tensor products. It should reduce some of the tedium of tasks involving fundamental crystals *A*, *B*, *C*, and *D*. Through object-oriented design, the code for the engine will be flexible enough to accommodate many improvements in the future.

One unresolved issue is the ability of the command-line interface to handle input errors. If given an incorrectly formatted command, *cg_console* will not always recover and present the user with a new prompt. In some cases it will even crash. The next programmer given ownership of the code should work on this problem.

Littelmann paths have not yet been implemented, although the engine contains a “stub” class *vertex_lp* to accommodate them. This method of calculating crystal graph structure will involve more vector operations than the current decision-based algorithms. The next programmer may wish to use a linear algebra code library to make implementing Littelmann paths easier.

Finally, one long-term goal of the project might be to implement other root systems in addition to the current four. Another potential long-term goal is to create a graphical user interface that produces visualizations of graph structure.

Bibliography

Kashiwara, Masaki. On Crystal Bases. Proceedings of the 1994 Annual Seminar of the Canadian Mathematical Society. Providence: American Mathematical Society, 1995.

Kashiwara, Masaki and Nakashima, Toshiki. "Crystal Graphs for Representations of the q -Analogue of Classical Lie Algebras". Journal of Algebra 165 (1994): 295-345.

Error Messages

- Error Text: undefined (first, second) ID
Identifiers: UNDEF_ID, UNDEF_ID_A, UNDEF_ID_B
Explanation: The given identifier is not associated with a vertex
- Error Text: invalid (first, second) ID
Identifiers: INVALID_ID, INVALID_ID_A, INVALID_ID_B
Explanation: A vertex identifier in the command is syntactically invalid (e.g., the identifier is not a single letter or the dot)
- Error Text: undefined edge
Identifier: UNDEF_EDGE
Explanation: No appropriate edge (in the correct direction) of the given color is associated with the given vertex
- Error Text: invalid edge
Identifier: INVALID_EDGE
Explanation: The given edge color is invalid under the given vertex's root system
- Error Text: invalid specification for construction
Identifier: INVALID_SPEC
Explanation: Issued in response to a *make* command; the family specifier is invalid or a vertex with the given number and bar status does not exist in the graph with the given family and rank
- Error Text: root system mismatch
Identifier: RS_MISMATCH
Explanation: Issued in response to a *tensor* command; not all of the vertices in the tensor list match the given family and rank
- Error Text: (no error)
Identifier: NO_ERROR
Explanation: Placeholder value that indicates no error occurred

Revision History

- 0.0
 - First executable build
 - Most commands nonfunctional or disabled
- 0.1
 - First functional build
 - All commands except *tensor* stable (albeit error-prone)
- 0.2
 - Bounds check of edge color applied to all vertex navigation commands
 - Commands *big_e* and *big_f* return the given vertex if it is at the beginning or end of the i-string
 - Invalid root systems B1, C1, and D1 no longer accepted as input
 - Miscellaneous fixes to vertex classes' navigation logic
- 0.5
 - '*' option available for selected commands
 - Non-abstract vertex classes provide self-copying routines
 - Redundancy in class *tensor_vertex* constructors reduced
 - Commands *e*, *f*, *big_e*, and *big_f* "move" vertices instead of creating new ones
 - Class *lp_vertex* directly derived from *vertex*
 - Class *simple_vertex* holds common code for singleton vertex classes
 - Function *ibounds* function moved from class *tensor_vertex* to class *cg_engine*
 - All input bounds checking moved from vertex classes to class *cg_engine*
- 1.0
 - Command *tensor* implemented and working
 - User interface slightly more forgiving to incorrect input
- 1.1
 - Functions *big_e* and *big_f* of *tensor_vertex* rewritten to be more time-efficient
 - "Vertex ID" added to information shown in output for command *display*

Source Code Reference: Classes

Name: cg_console

Purpose: Provides a console-based user interface with the crystal graph engine

Relations: cg_console, ostream, istream

Functions: cg_console(istream& pin, ostream& pout)
Constructor; parameters are input and output streams for console
~cg_console()
Destructor; deallocates the session cg_engine
bool IsReady() const
Returns true only if the session cg_engine initialized correctly
void Go()
Instructs the console to handle commands from the input stream
(private) string GetErrorString(cg_error code) const
Retrieves the error string associated with an error identifier
(private) string MakeLower(string s) const
Convert all letters in a string to lower case

Data: (private) istream& in
Reference to the input stream for the console
(private) ostream& out
Reference to the output stream for the console
(private) cg_engine* cge
Pointer to the cg_engine object for the session

Source Code Reference: Classes (*Continued*)

Name: cg_engine

Purpose: Executes commands from cg_console upon vertices

Relations: vertex

Functions: cg_engine()
Constructor; initializes variable slot table

~cg_engine()
Destructor; deallocates any present vertices

cg_error big_e(int color, char id, string& disp)
cg_error big_f(int color, char id, string& disp)
cg_error e(int color, char id, string& disp)
cg_error f(int color, char id, string& disp)
Vertex recalculation command handlers; recalculate vertex *id* and place printable result in *disp*

cg_error copy(char dest, char src, string& disp)
Copy command handler; copies vertex *src* to vertex *dest* and places a printable description of it in *disp*

cg_error display(char id, string& disp)
Display command handler; places a printable description of vertex *id* into *disp*

cg_error epsilon(int color, char id, string& result)
cg_error phi(int color, char id, string& result)
Epsilon and *Phi* command handlers; calculate distance to beginning or end (respectively) of vertex *id*'s *i*-string; place printable result in *result*

cg_error erase(char id)
Erase command handler; deallocates vertex *id*

cg_error make(char id, char family, int rank, int loc, bool bar, string& disp)
Make command handler; see "User Interface: List of Commands" for parameter explanations; places description of created vertex in *disp*

Source Code Reference: Classes (*Continued*)

Functions: `cg_error tensor(char id, char family, int rank, int numfacts, string factors, string& disp)`
(continued) `Tensor` command handler; see “User Interface: List of Commands” for parameter explanations; places description of created vertex in `disp`

`(private) inline bool cid(char c, int& i)`
 If a variable identifier `c` is acceptable, converts it to a variable slot `i`

`(private) inline bool idc(int i, char& c)`
 If a variable slot `i` is acceptable, converts it to a variable identifier `c`

`(private) string UpdateDot(int i)`
 Copies the vertex in variable slot `i` into the return register

Source Code Reference: Classes (*continued*)

`(private) inline bool BoundsCheck(int i, int color) const`
 Performs a bounds check on `color`; returns true if `color` is acceptable within the root system of the vertex in slot `i`

Data: `(private) vertex* slots[27]`
 Pointers for the 26 possible user-defined vertex variables (slots 1-26) and the return register (slot 0)

Source Code Reference: Classes (*Continued*)

Name: vertex

Purpose: Virtual base class for all vertices

Relations: (none)

Functions: vertex(v_family family, int rank)
 Constructor; creates a vertex of given *family* and *rank*

vertex(const vertex& copy)
 Copy constructor

~vertex()
 Destructor; deallocates Cartan matrix

vertex& operator=(const vertex& copy)
 Assignment operator

virtual bool big_e(int color)=0

virtual bool big_f(int color)=0

virtual bool e(int color)=0

virtual bool f(int color)=0
 Recalculate vertex along *color*-string

virtual int epsilon(int color)=0

virtual int phi(int color)=0
 Return distance to beginning or end (respectively) of *color*-string

virtual string GetDescription() const=0
 Returns a printable description of the vertex's properties

root_system GetRootSystem() const;
 Returns the root system of the vertex

virtual vertex* MakeCopy() const=0;
 Return a copy of the vertex

(*protected*) virtual void FillCartanMatrix()
 Calculate and fill the vertex's Cartan matrix; not yet implemented

Data: (*protected*) root_system rs
 Specifies the family and rank of the vertex

Source Code Reference: Classes (*continued*)

Source Code Reference: Classes (*Continued*)

Name: simple_vertex
 Purpose: Virtual base class for all singleton vertices
 Relations: Derived from *vertex* with public inheritance
 Functions: simple_vertex(v_family family, int rank, int ploc, bool pbar)
 Constructor; makes a vertex with given *family* and *rank* at location
 ploc and *pbar*
 simple_vertex(const simple_vertex& copy)
 Copy constructor
 string GetDescription() const
 string ShortDescription() const
 (Actualizations of *vertex* virtual functions)
 Data: (protected) int loc
 Location “number” of vertex in crystal graph
 (protected) Int bar
 Presence of “bar” in vertex’s identifying location

Name: tensor_vertex
 Purpose: Represents vertices that are the results of tensor operations
 Relations: Derived from *vertex* with public inheritance
 Functions: tensor_vertex(v_family family, int rank, int porder, vertex* const pfactors[])
 Constructor; makes a vertex with given *family* and *rank* out of
 porder vertices pointed to by *pfactors*[]
 tensor_vertex(const tensor_vertex& copy)
 Copy constructor
 ~tensor_vertex()
 Destructor; deallocates stored factors in tensor product
 (private) bool Backward(int color, int& k, int& _epsilon)
 (private) bool Forward(int color, int& k, int& _phi)
 Algorithms used to calculate *e*, *f*, *epsilon* and *phi*

Source Code Reference: Classes (*Continued*)

Functions: virtual bool big_e(int color)=0
 (*Continued*) virtual bool big_f(int color)=0
 virtual bool e(int color)=0
 virtual bool f(int color)=0
 virtual int epsilon(int color)=0
 virtual int phi(int color)=0
 string GetDescription() const
 string ShortDescription() const
 (Actualizations of *vertex* virtual functions)

Data: (*private*) int order
 Number of factors in the vertex
 (*private*) vertex** factors
 Array of pointers to factors in the vertex

Names: vertex_a, vertex_b, vertex_c, vertex_d

Purpose: Represent singleton vertices (e.g., those not the product of tensors)

Functions: vertex_x(int rank, int ploc, bool pbar) (*where x is 'a', 'b', 'c', or 'd'*)
 Creates a vertex of family *x* and *rank* at location *ploc* and bar status *pbar*
 virtual bool big_e(int color)=0
 virtual bool big_f(int color)=0
 virtual bool e(int color)=0
 virtual bool f(int color)=0
 virtual int epsilon(int color)=0
 virtual int phi(int color)=0
 vertex* MakeCopy() const
 (*protected*) void FillCartanMatrix()
 (Actualizations of *vertex* virtual functions)

Source Code Reference: Additional Data Types

```
typedef enum {NO_ERROR, UNDEF_ID, UNDEF_ID_A, UNDEF_ID_B, INVALID_ID,  
INVALID_ID_A, INVALID_ID_B, UNDEF_EDGE, INVALID_EDGE, INVALID_SPEC,  
RS_MISMATCH} cg_error;
```

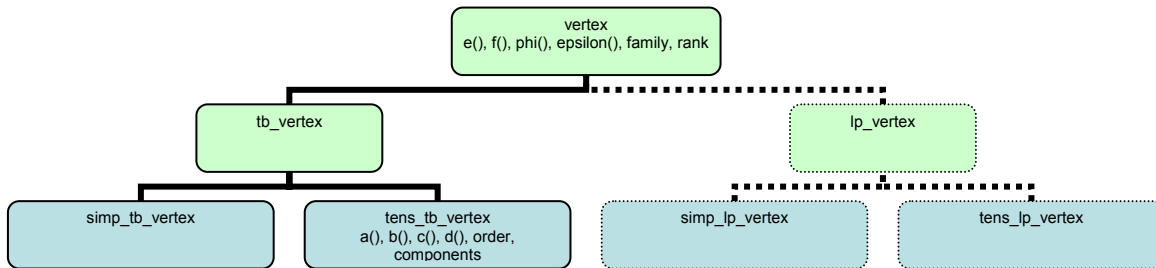
Error specifier used in *cg_engine* and *cg_console* classes

```
typedef enum {FAMILY_A=65, FAMILY_B=66, FAMILY_C=67, FAMILY_D=68} v_family;
```

Crystal graph family specifier used in *vertex* classes

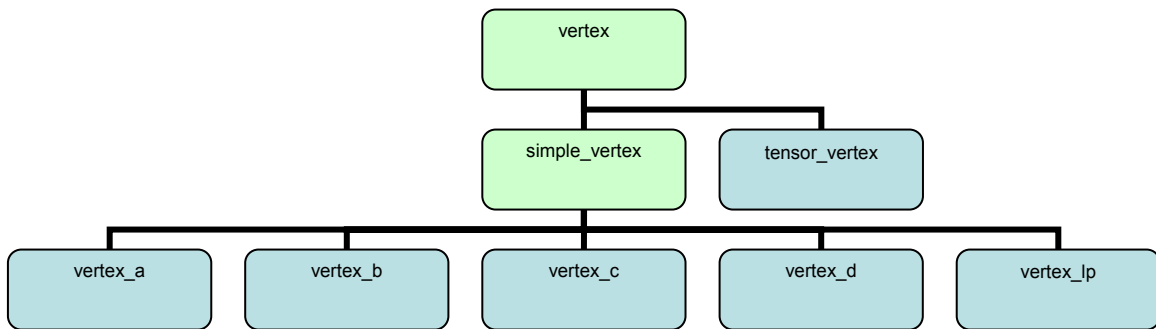
```
struct root_system{int rank; int** cartan_matrix; v_family family;};
```

Structure holding root system information used in *vertex* classes

Vertex Class Hierarchy, Revision 1 (*Obsolete*)

Notes

- `vertex`, `tb_vertex`, and `lp_vertex` are pure virtual base classes.
- `lp_vertex`, `simp_lp_vertex`, and `tens_lp_vertex` will be implemented in the next phase of the project.
- `operator*` will be implemented externally for each valid combination of operands.
- `vertex` functions will serve as the primary interface with the “outside world”.

Vertex Class Hierarchy, Revision 2 (*Obsolete*)

Graph-Related Class Members

Class Name	Methods	Data Members
Vertex	Graph Methods: virtual vertex e() $=0$; virtual vertex f() $=0$; virtual vertex phi() $=0$; virtual vertex epsilon() $=0$;	root_system rs;
simple_vertex		
tensor_vertex	(Graph Methods)	
vertex_a	(Graph Methods)	
vertex_b	(Graph Methods)	
vertex_c	(Graph Methods)	
vertex_d	(Graph Methods)	
vertex_lp	(Graph Methods)	
root_system		char family; int rank; matrix cartan_matrix;

Interface Commands, Revision 1 (*Obsolete*)

- Command Name: help
Parameters: none
Result: displays a list of available commands
- Command Name: make $f r$
Parameters: f : family specifier ('a', 'b', 'c', 'd')
 r : rank specifier (positive integer)
Result: returns the highest weight vertex of the graph of family
- Command Name: copy $d s$
Parameters: d, s : vertices
Result: copies vertex s into d , creating d if necessary
- Command Name: tens $a b$
Parameters: a, b : vertices of the same root system
Result: returns the highest weight vector in the graph of the tensor product of the graphs specified by a and b
- Command Name: disp a
Parameters: a : vertex or '*'
Result: displays vertex a ; or if a is '*' then all declared vertices are displayed
- Command Name: eras a
Parameters: a : vertex or '*'
Result: erases vertex a ; or if a is '*' then all declared vertices are erased

Interface Commands, Revision 2 (*Obsolete*)

Revision 2: 14 OCT 2003

- Command Name: `big_e i a`
Parameters: `i`: color (positive integer)
`a`: vertex
Result: returns the first vertex from `a` in the `i`-string
- Command Name: `big_f i a`
Parameters: `i`: color (positive integer)
`a`: vertex
Result: returns the last vertex from `a` in the `i`-string
- Command Name: `copy d s`
Parameters: `d, s`: vertices
Result: copies vertex `s` into `d`, creating `d` if necessary
- Command Name: `display a`
Parameters: `a`: vertex or `**`
Result: displays vertex `a`; or if `a` is `**` then all declared vertices are displayed
- Command Name: `e i a`
Parameters: `i`: color (positive integer)
`a`: vertex
Result: returns the previous vertex from `a` in the `i`-string
- Command Name: `epsilon i a`
Parameters: `i`: color (positive integer)
`a`: vertex
Result: returns the number of steps to the beginning of the `i`-string
- Command Name: `erase a`
Parameters: `a`: vertex or `**`

Result: erases vertex a ; or if a is $**$ then all declared vertices are erased

Command Name: $f i a$

Parameters: i : color (positive integer)

a : vertex

Result: returns the next vertex from a in the i -string

Command Name: $help$

Parameters: none

Result: displays a list of available commands

Command Name: $make a f r n b$

Parameters: a : name for vertex

f : family specifier ('a', 'b', 'c', 'd')

r : rank specifier (positive integer)

n : "number" associated with the vertex (positive integer)

b : whether the representation of the vertex has a bar (0 or 1)

Result: returns the specified vertex of the specified graph

Command Name: $phi i a$

Parameters: i : color (positive integer)

a : vertex

Result: returns the number of steps to the end of the i -string

Command Name: $tensor a b$

Parameters: a, b : vertices of the same root system

Result: returns the highest weight vector in the graph of the tensor product of the graphs specified by a and b