BGL and the Generic Programming Process

Jaakko Järvi
jarvi@cs.tamu.edu

August 31, 2005

Generic Programming Process

- Identify useful and efficient algorithms
- Find their generic representation
  - Categorize functionality of some of these algorithms
  - What do they need to have in order to work *in principle*
- Derive a set of (minimal) requirements that allow these algorithms to run (efficiently)
  - Now categorize these algorithms and their requirements
  - Are there overlaps, similarities?
- Construct a framework based on classifications and requirements
  - Now realize this as a software library
- Let's look at parts of this process with an example in the context of graph algorithms.

Process

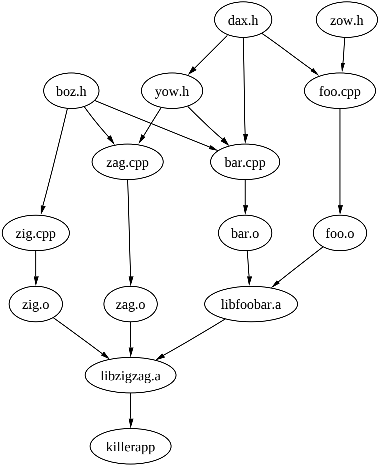
- Problem domain: dependencies, networks, graphs
- Solution domain: procedures for ordering and searching, data structures for graphs
- Identify a collection of efficient algorithms.
- Categorize their functionality, what do they need to have in order to work.
  - Look for commonality in the control flow of the procedures
  - Identify differences in data representation and in the actions

Process

- Problem domain: dependencies, networks, graphs
- Solution domain: procedures for ordering and searching, data structures for graphs
- Identify a collection of efficient algorithms.
- Categorize their functionality, what do they need to have in order to work.
  - Look for commonality in the control flow of the procedures
  - Identify differences in data representation and in the actions

Problem domain: Makefile dependencies

$a \longrightarrow b$ means a is used by b and thus must be built before b.



Order to build targets in a makefile

Intuition: if build target u depends on v, build v before u.

```
struct build_target {
   string name;
   bool visited;
   vector<build_target*> depends_on;
};
vector<build_target*> order;

void sort_build(build_target* u) {
   u→ visited = true;
   for (int i=0; i != u→ depends_on.size(); ++i){
      build_target* v = u→ depends_on[i];
      if (! v→ visited)
         sort_build(v);
   }
   order.push_back(u);
}
```

Order to build targets in a makefile

Intuition: if build target u is used by v, build u before v.

```
struct build_target {
    string name;
    bool visited;
    vector<build_target*> depends_on;
};
vector<build_target*> order;

void sort_build(build_target* u) {
    u→ visited = true;
    for (int i=0; i != u→ depends_on.size(); ++i){
        build_target* v = u→ depends_on[i];
        if (! v→ visited)
            sort_build(v);
    }
    order.push_back(u);
}
```

Order to build targets in a makefile

Intuition: if build target u is used by v, build u before v.

```
struct build_target {
   string name;
   bool visited;
   vector<build_target*> depends_on;
};
vector<build_target*> order;

void sort_build(build_target* u) {
   u→ visited = true;
   for (int i=0; i != u→ depends_on.size(); ++i){
      build_target* v = u→ depends_on[i];
      if (! v→ visited)
         sort_build(v);
   }
   order.push_back(u);
}
```

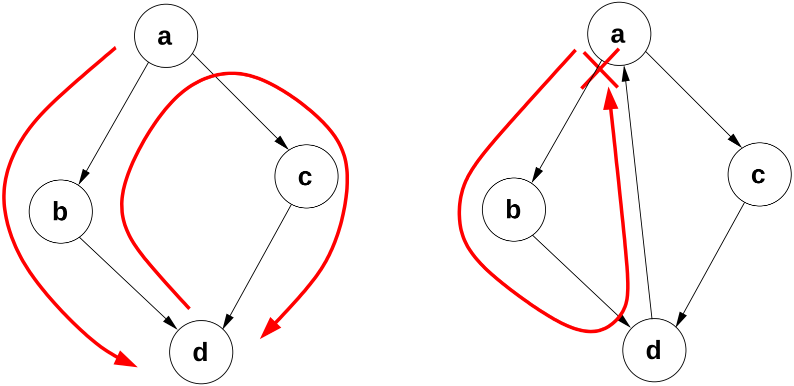
Order to build targets in a makefile

Intuition: if build target u is used by v, build u before v.

```
struct build_target {
  string name;
  bool visited;
  vector<build_target*> depends_on;
};
vector<build_target*> order;

void sort_build(build_target* u) {
  u→ visited = true;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (! v→ visited)
      sort_build(v);
  }
  order.push_back(u);
}
```

Detect cycles in a makefile

Intuition: follow dependencies and check whether you ever find a dependency such as $d \to a$ where a is an "ancestor" of d.



Detect cycles in a makefile

```
enum Status { untouched, visiting_children, finished };
struct build_target {
  string name;
  Status status;
  vector<build_target*> depends_on;
};

bool has_cycle(build_target* u) {
  u→ status = visiting_children;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (v→ status == untouched) {
      if (has_cycle(v)) return true;
    } else if (v→ status == visiting_children)
      return true;
  }
  u→ status = finished;
  return false;
}
```

Detect cycles in a makefile

```
enum Status { untouched, visiting_children, finished };
struct build_target {
  string name;
  Status status;
  vector<build_target*> depends_on;
};

bool has_cycle(build_target* u) {
  u→ status = visiting_children;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (v→ status == untouched) {
      if (has_cycle(v)) return true;
    } else if (v→ status == visiting_children)
      return true;
  }
  u→ status = finished;
  return false;
}
```

Detect cycles in a makefile

```
enum Status { untouched, visiting_children, finished };
struct build_target {
  string name;
  Status status;
  vector<build_target*> depends_on;
};

bool has_cycle(build_target* u) {
  u→ status = visiting_children;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (v→ status == untouched) {
      if (has_cycle(v)) return true;
    } else if (v→ status == visiting_children)
      return true;
  }
  u→ status = finished;
  return false;
}
```

Detect cycles in a makefile

```
enum Status { untouched, visiting_children, finished };
struct build_target {
  string name;
  Status status;
  vector<build_target*> depends_on;
};

bool has_cycle(build_target* u) {
  u→ status = visiting_children;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (v→ status == untouched) {
      if (has_cycle(v)) return true;
    } else if (v→ status == visiting_children)
      return true;
  }
  u→ status = finished;
  return false;
}
```

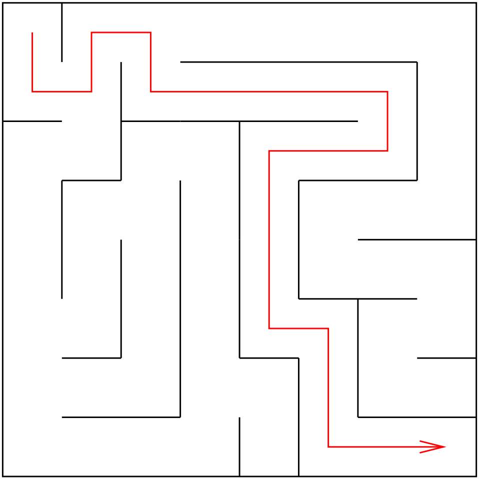
Detect cycles in a makefile

```
enum Status { untouched, visiting_children, finished };
struct build_target {
   string name;
   Status status;
   vector<build_target*> depends_on;
};

bool has_cycle(build_target* u) {
   u→ status = visiting_children;
   for (int i=0; i != u→ depends_on.size(); ++i){
      build_target* v = u→ depends_on[i];
      if (v→ status == untouched) {
         if (has_cycle(v)) return true;
      } else if (v→ status == visiting_children)
         return true;
   }
   u→ status = finished;
   return false;
}
```

Detect cycles in a makefile

```
enum Status { untouched, visiting_children, finished };
struct build_target {
  string name;
  Status status;
  vector<build_target*> depends_on;
};

bool has_cycle(build_target* u) {
  u→ status = visiting_children;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (v→ status == untouched) {
      if (has_cycle(v)) return true;
    } else if (v→ status == visiting_children)
      return true;
  }
  u→ status = finished;
  return false;
}
```

Problem domain: path through a maze



Find a path out of a maze

Intuition: be systematic, and drop bread crumbs.

```
struct room { bool door[4]; }; // Door to the north, south, east,
    west?
int x_offset[] = { 0, 0, 1, -1 }, y_offset[] = { -1, 1, 0, 0 };
multi_array<room, 2> maze(extents[n][n]);
multi_array<bool, 2> visited(extents[n][n]);
stack< pair<int,int> > path;

bool find_path(int x, int y) {
  visited[x][y] = true;
  path.push(make_pair(x,y));
  if (x == n-1 && y == n-1) return true;
  for (int i=0; i != 4; ++i)
    if (maze[x][y].door[i]) {
      int new_x = x + x_offset[i];
      int new_y = y + y_offset[i];
      if (! visited[new_x][new_y])
        if (find_path(new_x, new_y))
          return true;
    }
  path.pop();
  return false;
}
```

```
struct room { bool door[4]; }; // Door to the north, south, east,
      west?
int x_offset[] = { 0, 0, 1, -1 }, y_offset[] = { -1, 1, 0, 0 };
multi_array<room, 2> maze(extents[n][n]);
multi_array<bool, 2> visited(extents[n][n]);
stack< pair<int,int> > path;
bool find_path(int x, int y) {
  visited[x][y] = true;
  path.push(make_pair(x,y));
  if (x == n-1 && y == n-1) return true;
  for (int i=0; i != 4; ++i)
    if (maze[x][y].door[i]) {
      int new_x = x + x_offset[i];
      int new_y = y + y_offset[i];
      if (! visited[new_x][new_y])
        if (find_path(new_x, new_y))
          return true;
    }
  path.pop();
  return false;
}
```

```
struct room { bool door[4]; }; // Door to the north, south, east,
      west?
int x_offset[] = { 0, 0, 1, -1 }, y_offset[] = { -1, 1, 0, 0 };
multi_array<room, 2> maze(extents[n][n]);
multi_array<bool, 2> visited(extents[n][n]);
stack< pair<int,int> > path;

bool find_path(int x, int y) {
   visited[x][y] = true;
   path.push(make_pair(x,y));
   if (x == n-1 && y == n-1) return true;
   for (int i=0; i != 4; ++i)
      if (maze[x][y].door[i]) {
         int new_x = x + x_offset[i];
         int new_y = y + y_offset[i];
         if (! visited[new_x][new_y])
            if (find_path(new_x, new_y))
               return true;
      }
   path.pop();
   return false;
}
```

```
struct room { bool door[4]; }; // Door to the north, south, east,
      west?
int x_offset[] = { 0, 0, 1, −1 }, y_offset[] = { −1, 1, 0, 0 };
multi_array<room, 2> maze(extents[n][n]);
multi_array<bool, 2> visited(extents[n][n]);
stack< pair<int,int> > path;

bool find_path(int x, int y) {
  visited[x][y] = true;
  path.push(make_pair(x,y));
  if (x == n−1 && y == n−1) return true;
  for (int i=0; i != 4; ++i)
    if (maze[x][y].door[i]) {
      int new_x = x + x_offset[i];
      int new_y = y + y_offset[i];
      if (! visited[new_x][new_y])
        if (find_path(new_x, new_y))
          return true;
    }
  path.pop();
  return false;
}
```

Process

- Problem domain: dependencies, networks, graphs
- Solution domain: procedures for ordering and searching, data structures for graphs
- Identify a collection of efficient algorithms.
- Categorize their functionality, what do they need to have in order to work.
  - Look for commonality in the control flow of the procedures
  - Identify differences in data representation and in the actions

Commonality?

```
void sort_build(build_target* u) {
  u→ visited = true;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (! v→ visited)
      sort_build(v);
  }
  order.push_back(u);
}
```

```
bool has_cycle(build_target* u) {
  u→ status = visiting_children;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (v→ status == untouched) {
      if (has_cycle(v)) return true;
    } else if (v→ status == visiting_children)
      return true;
  }
  u→ status = finished;
  return false;
}
```

```
bool find_path(int x, int y) {
  visited[x][y] = true;
  path.push(make_pair(x,y));
  if (x == n−1 && y == n−1) return true;
  for (int i=0; i != 4; ++i)
    if (maze[x][y].door[i]) {
      int new_x = x + x_offset[i];
      int new_y = y + y_offset[i];
      if (! visited[new_x][new_y])
        if (find_path(new_x, new_y))
          return true;
    }
  path.pop();
  return false;
}
```

Commonality: Mark places that have been visited

```
void sort_build(build_target* u) {
  u→ visited = true;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (! v→ visited)
      sort_build(v);
  }
  order.push_back(u);
}
```

```
bool has_cycle(build_target* u) {
  u→ status = visiting_children;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (v→ status == untouched) {
      if (has_cycle(v)) return true;
    } else if (v→ status == visiting_children)
      return true;
  }
  u→ status = finished;
  return false;
}
```

```
bool find_path(int x, int y) {
  visited[x][y] = true;
  path.push(make_pair(x,y));
  if (x == n−1 && y == n−1) return true;
  for (int i=0; i != 4; ++i)
    if (maze[x][y].door[i]) {
      int new_x = x + x_offset[i];
      int new_y = y + y_offset[i];
      if (! visited[new_x][new_y])
        if (find_path(new_x, new_y))
          return true;
    }
  path.pop();
  return false;
}
```

Commonality: Visit adjacent locations

```
void sort_build(build_target* u) {
  u→ visited = true;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (! v→ visited)
      sort_build(v);
  }
  order.push_back(u);
}
```

```
bool has_cycle(build_target* u) {
  u→ status = visiting_children;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (v→ status == untouched) {
      if (has_cycle(v)) return true;
    } else if (v→ status == visiting_children)
      return true;
  }
  u→ status = finished;
  return false;
}
```

```
bool find_path(int x, int y) {
  visited[x][y] = true;
  path.push(make_pair(x,y));
  if (x == n−1 && y == n−1) return true;
  for (int i=0; i != 4; ++i)
    if (maze[x][y].door[i]) {
      int new_x = x + x_offset[i];
      int new_y = y + y_offset[i];
      if (! visited[new_x][new_y])
        if (find_path(new_x, new_y))
          return true;
    }
  path.pop();
  return false;
}
```

Commonality: Process neighbors if not yet visited

```
void sort_build(build_target* u) {
  u→ visited = true;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (! v→ visited)
      sort_build(v);
  }
  order.push_back(u);
}
```

```
bool has_cycle(build_target* u) {
  u→ status = visiting_children;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (v→ status == untouched) {
      if (has_cycle(v)) return true;
    } else if (v→ status == visiting_children)
      return true;
  }
  u→ status = finished;
  return false;
}
```

```
bool find_path(int x, int y) {
  visited[x][y] = true;
  path.push(make_pair(x,y));
  if (x == n−1 && y == n−1) return true;
  for (int i=0; i != 4; ++i)
    if (maze[x][y].door[i]) {
      int new_x = x + x_offset[i];
      int new_y = y + y_offset[i];
      if (! visited[new_x][new_y])
        if (find_path(new_x, new_y))
          return true;
    }
  path.pop();
  return false;
}
```

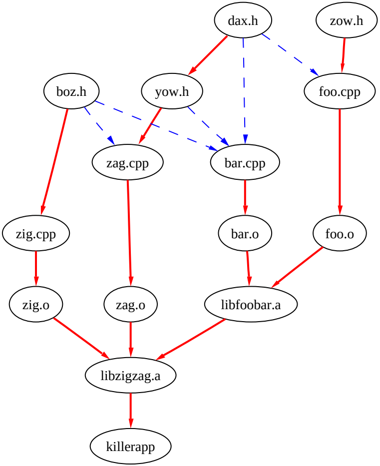
Commonality: Action after neighbors are explored

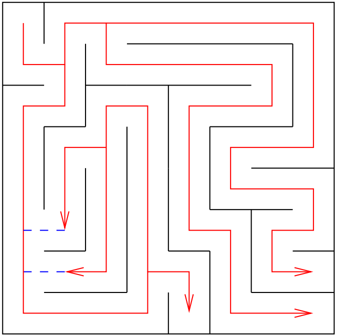
```
void sort_build(build_target* u) {
  u→ visited = true;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (! v→ visited)
      sort_build(v);
  }
  order.push_back(u);
}
```

```
bool has_cycle(build_target* u) {
  u→ status = visiting_children;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (v→ status == untouched) {
      if (has_cycle(v)) return true;
    } else if (v→ status == visiting_children)
      return true;
  }
  u→ status = finished;
  return false;
}
```

```
bool find_path(int x, int y) {
  visited[x][y] = true;
  path.push(make_pair(x,y));
  if (x == n−1 && y == n−1) return true;
  for (int i=0; i != 4; ++i)
    if (maze[x][y].door[i]) {
      int new_x = x + x_offset[i];
      int new_y = y + y_offset[i];
      if (! visited[new_x][new_y])
        if (find_path(new_x, new_y))
          return true;
    }
  path.pop();
  return false;
}
```

Commonality: Depth First Search (DFS)





DFS tree edges are in red and non-tree in blue.

Process

- Problem domain: dependencies, networks, graphs
- Solution domain: procedures for ordering and searching, data structures for graphs
- Identify a collection of efficient algorithms.
- Categorize their functionality, what do they need to have in order to work.
  - Look for commonality in the control flow of the procedures
  - Identify differences in data representation and in the actions

Differences?

```
void sort_build(build_target* u) {
  u→ visited = true;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (! v→ visited)
      sort_build(v);
  }
  order.push_back(u);
}
```

```
bool has_cycle(build_target* u) {
  u→ status = visiting_children;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (v→ status == untouched) {
      if (has_cycle(v)) return true;
    } else if (v→ status == visiting_children)
      return true;
  }
  u→ status = finished;
  return false;
}
```

```
bool find_path(int x, int y) {
  visited[x][y] = true;
  path.push(make_pair(x,y));
  if (x == n−1 && y == n−1) return true;
  for (int i=0; i != 4; ++i)
    if (maze[x][y].door[i]) {
      int new_x = x + x_offset[i];
      int new_y = y + y_offset[i];
      if (! visited[new_x][new_y])
        if (find_path(new_x, new_y))
          return true;
    }
  path.pop();
  return false;
}
```

Difference: Representation of Nodes

```
void sort_build(build_target* u) {
  u→ visited = true;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (! v→ visited)
      sort_build(v);
  }
  order.push_back(u);
}
```

```
bool has_cycle(build_target* u) {
  u→ status = visiting_children;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (v→ status == untouched) {
      if (has_cycle(v)) return true;
    } else if (v→ status == visiting_children)
      return true;
  }
  u→ status = finished;
  return false;
}
```

```
bool find_path(int x, int y) {
  visited[x][y] = true;
  path.push(make_pair(x,y));
  if (x == n−1 && y == n−1) return true;
  for (int i=0; i != 4; ++i)
    if (maze[x][y].door[i]) {
      int new_x = x + x_offset[i];
      int new_y = y + y_offset[i];
      if (! visited[new_x][new_y])
        if (find_path(new_x, new_y))
          return true;
    }
  path.pop();
  return false;
}
```

Difference: Representation of Connections

```
void sort_build(build_target* u) {
  u→ visited = true;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (! v→ visited)
      sort_build(v);
  }
  order.push_back(u);
}
```

```
bool has_cycle(build_target* u) {
  u→ status = visiting_children;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (v→ status == untouched) {
      if (has_cycle(v)) return true;
    } else if (v→ status == visiting_children)
      return true;
  }
  u→ status = finished;
  return false;
}
```

```
bool find_path(int x, int y) {
  visited[x][y] = true;
  path.push(make_pair(x,y));
  if (x == n−1 && y == n−1) return true;
  for (int i=0; i != 4; ++i)
    if (maze[x][y].door[i]) {
      int new_x = x + x_offset[i];
      int new_y = y + y_offset[i];
      if (! visited[new_x][new_y])
        if (find_path(new_x, new_y))
          return true;
    }
  path.pop();
  return false;
}
```

Difference: Mapping node to marker

```
void sort_build(build_target* u) {
  u→ visited = true;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (! v→ visited)
      sort_build(v);
  }
  order.push_back(u);
}
```

```
bool has_cycle(build_target* u) {
  u→ status = visiting_children;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (v→ status == untouched) {
      if (has_cycle(v)) return true;
    } else if (v→ status == visiting_children)
      return true;
  }
  u→ status = finished;
  return false;
}
```

```
bool find_path(int x, int y) {
  visited[x][y] = true;
  path.push(make_pair(x,y));
  if (x == n−1 && y == n−1) return true;
  for (int i=0; i != 4; ++i)
    if (maze[x][y].door[i]) {
      int new_x = x + x_offset[i];
      int new_y = y + y_offset[i];
      if (! visited[new_x][new_y])
        if (find_path(new_x, new_y))
          return true;
    }
  path.pop();
  return false;
}
```

Difference: other actions

```
void sort_build(build_target* u) {
  u→ visited = true;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (! v→ visited)
      sort_build(v);
  }
  order.push_back(u);
}
```

```
bool has_cycle(build_target* u) {
  u→ status = visiting_children;
  for (int i=0; i != u→ depends_on.size(); ++i){
    build_target* v = u→ depends_on[i];
    if (v→ status == untouched)
      if (has_cycle(v)) return true;
    } else if (v→ status == visiting_children)
      return true;
  }
  u→ status = finished;
  return false;
}
```

```
bool find_path(int x, int y) {
  visited[x][y] = true;
  path.push(make_pair(x,y));
  if (x == n−1 && y == n−1) return true;
  for (int i=0; i != 4; ++i)
    if (maze[x][y].door[i]) {
      int new_x = x + x_offset[i];
      int new_y = y + y_offset[i];
      if (! visited[new_x][new_y])
        if (find_path(new_x, new_y))
          return true;
    }
  path.pop();
  return false;
}
```

Identify abstractions ("concepts")

- Use abstraction to smooth over the differences.
- Candidates:
  - graph with vertices and adjacent vertices (*Graph*)
  - mapping from vertex to visited marker (*Property Map*)
  - call-backs for events in the DFS (*DFS Visitor*)
- Define **concepts** that describe these absractions

Specify the *Graph* concept

Any type G is a model of the *Graph* concept if it satisfies the following requirements (v is an object of type G::vertex and g is an object of type G.)

Expression	Note
G::vertex	opaque handle to a vertex
G::adjacency_iterator	traverse adjacent vertices
adj(v, g)	returns pair<adjacent_iterator> in $O(1)$

- adj(v,g) returns in $O(1)$.
- The associated G::adjacency_iterator must model *Input Iterator* and its value_type must be G::vertex.
- The associated G::vertex must model *Copy Constructible* and *Equality Comparable*.

(*The use of member typedefs instead of traits classes is for brevity of code.*)

Specify the *Property Map* concept

Any type PMap is a model of *Property Map* concept if it satisfies the following requirements

Expression	Note
PMap::key	key type
PMap::value	value type
map[k]	returns value&

The interface for generic DFS

```
template <typename G, typename StatusMap>
void dfs(typename G::vertex u,
         const G& g,
         StatusMap smap, ...);
```

- Type requirements:
  - G models *Graph*
  - StatusMap models *Property Map*.
  - StatusMap::key is the same type as G::vertex.
  - StatusMap::value is Status.
- Requirements form a **contract** between client and the generic function. Concepts are used to express the requirements.

Other actions: DFS Visitor

- Encapsulate call-back event points in the algorithm.

  template <**class** G, **class** StatusMap, **class** DFSVisitor>
  void dfs(**typename** G::vertex u, **const** G& g,
  StatusMap smap, DFSVisitor vis);

- Requirements for a *DFS Visitor*:

  vis.start_vertex(u)
  vis.tree_edge(u,v)
  vis.back_edge(u,v)
  vis.forward_or_cross_edge(u,v)
  vis.finish_vertex(u)

Implementation of generic DFS

```
template <typename G, typename StatusMap, typename DFSVisitor>
void dfs(typename G::vertex u, const G& g, StatusMap status,
         DFSVisitor vis)
{
  status[u] = visiting_children;
  vis.start_vertex(u);
  typename G::adjacency_iterator first, last;
  for (tie(first, last) = adj(u, g); first != last; ++first) {
    typename G::vertex v = *first;
    if (status[v] == untouched) {
      vis.tree_edge(u,v);
      dfs(v, g, status, vis);
    } else if (status[v] == visiting_children)
      vis.back_edge(u,v);
    else
      vis.forward_or_cross_edge(u,v);
  }
  status[u] = finished;
  vis.finish_vertex(u);
}
```

Implementation of generic DFS

```
template <typename G, typename StatusMap, typename DFSVisitor>
void dfs(typename G::vertex u, const G& g, StatusMap status,
         DFSVisitor vis)
{
  status[u] = visiting_children;
  vis.start_vertex(u);
  typename G::adjacency_iterator first, last;
  for (tie(first, last) = adj(u, g); first != last; ++first) {
    typename G::vertex v = *first;
    if (status[v] == untouched) {
      vis.tree_edge(u,v);
      dfs(v, g, status, vis);
    } else if (status[v] == visiting_children)
      vis.back_edge(u,v);
    else
      vis.forward_or_cross_edge(u,v);
  }
  status[u] = finished;
  vis.finish_vertex(u);
}
```

Implementation of generic DFS

```
template <typename G, typename StatusMap, typename DFSVisitor>
void dfs(typename G::vertex u, const G& g, StatusMap status,
         DFSVisitor vis)
{
  status[u] = visiting_children;
  vis.start_vertex(u);
  typename G::adjacency_iterator first, last;
  for (tie(first, last) = adj(u, g); first != last; ++first) {
    typename G::vertex v = *first;
    if (status[v] == untouched) {
      vis.tree_edge(u,v);
      dfs(v, g, status, vis);
    } else if (status[v] == visiting_children)
      vis.back_edge(u,v);
    else
      vis.forward_or_cross_edge(u,v);
  }
  status[u] = finished;
  vis.finish_vertex(u);
}
```

Implementation of generic DFS

```
template <typename G, typename StatusMap, typename DFSVisitor>
void dfs(typename G::vertex u, const G& g, StatusMap status,
         DFSVisitor vis)
{
  status[u] = visiting_children;
  vis.start_vertex(u);
  typename G::adjacency_iterator first, last;
  for (tie(first, last) = adj(u, g); first != last; ++first) {
    typename G::vertex v = *first;
    if (status[v] == untouched) {
      vis.tree_edge(u,v);
      dfs(v, g, status, vis);
    } else if (status[v] == visiting_children)
      vis.back_edge(u,v);
    else
      vis.forward_or_cross_edge(u,v);
  }
  status[u] = finished;
  vis.finish_vertex(u);
}
```

Graph for Makefile Targets

```
class makefile_graph {
public:
  typedef build_target* vertex;
  typedef vector<build_target*>::iterator adjacency_iterator;
};

pair<vector<build_target*>::iterator,
     vector<build_target*>::iterator>
adj(build_target* u, const makefile_graph& g) {
  return make_pair(u→ depends_on.begin(), u→ depends_on.
        end());
}
```

- Thus makefile_graph is a model of the *Graph* concept.

Status Property Map

```
class target_status_map {
public:
  typedef build_target* key;
  typedef Status value;
  Status& operator[](build_target* u) const {
    return u→ status;
  }
};
```

DFS Visitor for Makefile dependencies and cycles

```
struct cycle { };
class makefile_dfs_visitor : public default_dfs_visitor {
public:
  makefile_dfs_visitor(vector<build_target*>* order)
    : order(order) { }

  void finish_vertex(build_target* u) {
    order→ push_back(u);
  }
  void back_edge(build_target* u, build_target* v) {
    throw cycle();
  }

  vector<build_target*>* order;
};
```

Use of DFS for dependency sorting and cycle detection

```
makefile_graph g;
target_status_map status_map;
makefile_dfs_visitor vis(&order);
try {
   for (int i=0; i != num_targets; ++i)
      if (target[i]→ status == untouched)
         dfs(target[i], g, status_map, vis);
} catch (cycle) {
   cout << "There was a cycle" << endl;
}
```

Graph for a Maze

```
class maze_adj_iter;
struct maze_graph {
   typedef pair<int,int> vertex;
   typedef maze_adj_iter adjacency_iterator;
};
class maze_adj_iter
      : public iterator_facade<maze_adj_iter, pair<int,int>,
            input_iterator_tag, pair<int,int> > {
   static int x_offset[], y_offset[];
   pair<int,int> pos; int i;
   void find_open_door();
public:
   maze_adj_iter() { }
   maze_adj_iter(const pair<int,int>& pos, int i)
      : pos(pos), i(i) { find_open_door(); }
   maze_adj_iter& operator++();
   pair<int,int> operator*() const;
   bool equal(const maze_adj_iter& iter) const;
};
```

Graph for a Maze (cont'd)

```
maze_adj_iter& maze_adj_iter::operator++()
    { ++i; find_open_door(); return *this; }

pair<int,int> maze_adj_iter::operator*() const
    { return make_pair(pos.first + x_offset[i],
                       pos.second + y_offset[i]); }

bool maze_adj_iter::equal(const maze_adj_iter& iter) const
    { return i == iter.i && pos == iter.pos; }

void maze_adj_iter::find_open_door() {
  while (i < 4 && !maze[pos.first][pos.second].door[i])
    ++i;
}

int maze_adj_iter::x_offset[] = { 0,0,1,-1 };
int maze_adj_iter::y_offset[] = { -1,1,0,0 };
```

Graph for a Maze (cont'd)

```
pair<maze_adj_iter, maze_adj_iter>
adj(const pair<int,int>& u, const maze_graph& g) {
  return make_pair(maze_adj_iter(u, 0),
                   maze_adj_iter(u, 4));
}
```

Thus maze_graph is a model of the *Graph* concept.

Status Property Map

```
class room_status_map {
  multi_array<Status, 2>* status;
public:
  typedef pair<int,int> key;
  typedef Status value;
  room_status_map(multi_array<Status, 2>* status)
    : status(status) { }
  Status& operator[](const pair<int,int>& u) const {
    return (*status)[u.first][u.second];
  }
};
```

Maze DFS Visitor

```
struct found_path { };
class maze_dfs_visitor : public default_dfs_visitor {
public:
  void start_vertex(pair<int,int> u) {
    path.push(u);
    if (u.first == n - 1 && u.second == n - 1)
      throw found_path();
  }
  void finish_vertex(pair<int,int> u) {
    path.pop();
  }
};
```

Use of DFS for Maze

```
maze_graph g;
room_status_map status_map(&status);
maze_dfs_visitor vis;
try {
  dfs(make_pair(0,0), g, status_map, vis);
} catch (found_path) {
  while (!path.empty()) {
    cout << "(" << path.top().first
          << "," << path.top().second << ")" << endl;
    path.pop();
  }
}
```