Algorithms Lab
Tutorial Session IV

First Steps with the BGL

Introduction

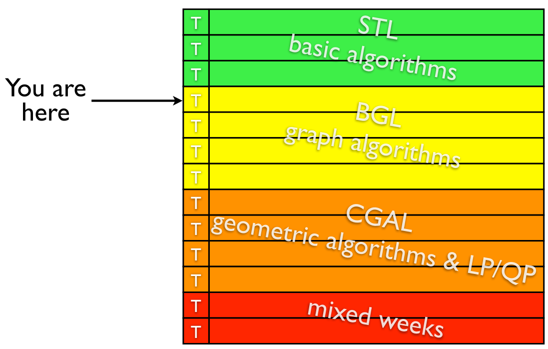
- boost is a free collection of portable C++ libraries
- boost graph library (BGL) is for graph algorithms:
  - shortest paths
  - flows
  - matchings
  - planar graphs
- BGL is highly customizable and extensible

  ⇒ generic programming

Introduction

- A short "tutorial" paper is available on moodle
  - "copy & paste tutorial"
- This presentation will also be available
- You will still need the BGL documentation a lot!
- Programming BGL requires some practice, be sure to start early!

Introduction
Timeline



Introduction
Timeline

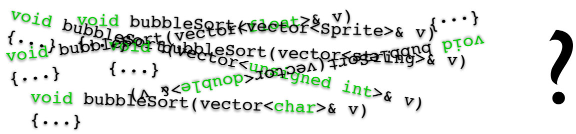
T	Week I: First steps with BGL
T	Week II: Flow-Week
T	Week III: Relax-Week (Matchings)
T	Week IV: Surprise-Week

Generic Programming

Generic Programming

```
void bubbleSort(vector<int>& v)
{
  for(int i = 0; i < v.size(); ++i)
    for(int j = 0; j < v.size()-1; ++j)
      if(v[j+1] < v[j])
        swap(v[j+1], v[j]);
}
```

- What if we want to sort floats, doubles, strings, etc.?



Generic Programming

```
template<typename T>  //'<' must be defined for T
void bubbleSort(vector<T>& v)
{
  for(int i = 0; i < v.size(); ++i)
    for(int j = 0; j < v.size()-1; ++j)
      if(v[j+1] < v[j])
        swap(v[j+1], v[j]);
}
```

- Use templates to abstract from internal type
- It is a best practice to specify our requirements for T
- What if we want to sort containers other than vector for which the '[]'-operator is sensibly defined?

Generic Programming

```
template<typename T>
void bubbleSort(T& v)
{
  for(int i = 0; i < v.size(); ++i)
    for(int j = 0; j < v.size()-1; ++j)
      if(v[j+1] < v[j])
        swap(v[j+1], v[j]);
}
```

- Advantage: generalized to all kinds of types with []-operator
- Disadvantage: requirements for T are not clear
  - Unfortunately, there is no way of enforcing an interface

  ⇒ Need to comment our function very carefully

  ⇒ Introduce 'concepts' as informal interfaces

Generic Programming

```
/*
    'RandomAccessContainer' concept:
    - t[i] must be a valid expression
    - t[i] must be a valid left-hand-side
    - t.size() must return the size of the container
    - '<' must be defined for all t[i], t[j]
*/

// T must be a model of the RandomAccess concept!
template<typename T>
void bubbleSort(T& v)
{
  for(int i = 0; i < v.size(); ++i)
    for(int j = 0; j < v.size()-1; ++j)
      if(v[j+1] < v[j])
        swap(v[j+1], v[j]);
}
```


Generic Programming

```
/*
    'ForwardIt' concept: ++, ==, !=
    'ReadWrite' concept: *it is valid lhs
    'Comparable' concept: *it1 < *it2
*/

// Iterator must model ForwardIt, ReadWrite, Comparable
template<typename Iterator>
void bubbleSort(Iterator begin, Iterator end)
{
  for(Iterator it = begin; it != end; ++it)
    for(Iterator it3 = begin, it2 = it3++; it3 != end;
        ++it2, ++it3)
      if(*it3 < *it2)
        swap(*it3, *it2);
}
```

⇒ STL style generic programming

Enter BGL

BGL
Generic Programming

- BGL aims at being extremely versatile and extensible
  - ⇒BGL algorithms should be applicable to any custom graphs that exhibit a minimal interface
  - ⇒generic programming
- BGL also provides graph implementations that model the different concepts

BGL
Graph Traversal Concepts

Concept	refines	types (traits)	expressions	runtime
Graph	--	vertex_descriptor directed_category edge_parallel_category traversal_category		
Incidence Graph	Graph	edge_descriptor out_edge_iterator degree_size_type	source(e,g) target(e,g) out_edges(v,g) out_degree(v,g)	O(1) O(1) O(1) $O(d^-(v))$
Bidirectional Graph	Incidence Graph	in_edge_iterator	in_edges(v,g) in_degree(v,g) degree(v,g)	O(1) $O(d^+(v))$
AdjacencyGraph	Graph	adjacency_iterator	adjacent_vertices(v,g)	O(1)
VertexListGraph	Graph	vertex_iterator vertices_size_type	vertices(g) num_vertices(g)	O(1) O(n)
EdgeListGraph	Graph	edge_descriptor edge_iterator edges_size_type	edges(g) source(e,g) target(e,g) num_edges(g)	O(1) O(1) O(1) O(m)
AdjacencyMatrix	Graph		edge(u,v,g)	O(1)

BGL
Graph Modification Concepts

Concept	refines	types (traits)	expressions	O
VertexMutable Graph	Graph		add_vertex(g) remove_vertex(v,g)	O(1*) O(m+n)
EdgeMutable Graph	Graph		clear_vertex(v,g) add_edge(u,v,g) remove_edge(u,v,g) remove_edge(e,g)	O(m+n) O(1*) O(m) O(m)
Mutable IncidenceGraph	IncidenceG, EdgeMutableG		remove_edge(eiter,g) remove_out_edge_if(u,p,g)	O(1) $O(d^-(v))$
MutableBidirectionalGraph	MutableIncidenceG,BidirectionalG		remove_edge(eiter,g) remove_out_edge_if(u,p,g)	O(1) $O(d^+(v))$
PropertyEdgeList Graph	Graph	property_map <G,T>::type proper... ::const_type	get(ptag,g) get(ptag,g,x) get(ptag,g,x,v)	O(1) O(1) O(1)
VertexMutable PropertyGraph	VertexMutableG, PropertyG		add_vertex(vp,g)	O(1*)
EdgeMutableGraph PropertyGraph	EdgeMutableG, PropertyG		add_edge(u,v,ep,g)	O(1*)

BGL
Workflow

1. Decide what algorithm you need
2. Check the documentation for the graph concepts required by the algorithm
3. Choose a suitable graph implementation
4. Set-up the graph and invoke algorithm

BGL
Workflow

1. Decide what algorithm you need
2. Check the documentation for the graph concepts required by the algorithm
3. Choose a suitable graph implementation
4. Set-up the graph and invoke algorithm

BGL
Example Algorithm

topological_sort

```
template <typename VertexListGraph, typename OutputIterator,
          typename P, typename T, typename R>
void topological_sort(VertexListGraph& g, OutputIterator result,
    const bgl_named_params<P, T, R>& params = all defaults)
```

The topological sort algorithm creates a linear ordering of the vertices such that if edge (u,v) appears in the graph, then v comes before u in the ordering. The graph must be a directed acyclic graph (DAG). The implementation consists mainly of a call to `depth_first_search()`.

Where Defined:

`boost/graph/topological_sort.hpp`

!! (note: this is not the whole truth!)

Parameters

IN: `VertexListGraph& g`

A directed acylic graph (DAG). The graph type must be a model of Vertex List Graph and Incidence Graph. If the graph is not a DAG then a `not_a_dag` exception will be thrown and the user should discard the contents of `result` range.
Python: The parameter is named `graph`.

OUT: `OutputIterator result`

The vertex descriptors of the graph will be output to the `result` output iterator in **reverse** topological order. The iterator type must model Output Iterator.
Python: This parameter is not used in Python. Instead, a Python `list` containing the vertices in topological order is returned.

BGL
Example Algorithm

Named Parameters

UTIL/OUT: `color_map(ColorMap color)`

This is used by the algorithm to keep track of its progress through the graph. The type `ColorMap` must be a model of Read/Write Property Map and its key type must be the graph's vertex descriptor type and the value type of the color map must model ColorValue.
Default: an iterator_property_map created from a `std::vector` of `default_color_type` of size `num_vertices(g)` and using the `i_map` for the index map.
Python: The color map must be a `vertex_color_map` for the graph.

IN: `vertex_index_map(VertexIndexMap i_map)`

This maps each vertex to an integer in the range `[0, num_vertices(g))`. This parameter is only necessary when the default color property map is used. The type `VertexIndexMap` must be a model of Readable Property Map. The value type of the map must be an integer type. The vertex descriptor type of the graph needs to be usable as the key type of the map.
Default: `get(vertex_index, g)` Note: if you use this default, make sure your graph has an internal `vertex_index` property. For example, `adjacenty_list` with `VertexList=listS` does not have an internal `vertex_index` property.
Python: Unsupported parameter.

Complexity

The time complexity is $O(V + E)$. !!

Example

BGL
Workflow

1. Decide what algorithm you need
2. Check the documentation for the graph concepts required by the algorithm
3. Choose a suitable graph implementation
4. Set-up the graph and invoke algorithm

BGL
some graph alternatives

type	Graph	models
BGL classes	adjacency_list	DefaultConstructible, CopyConstructible, Assignable, VertexListGraph, EdgeListGraph, IncidenceGraph, AdjacencyGraph, VertexMutableGraph, EdgeMutableGraph, BidirectionalGraph*, VertexMutablePropertyGraph*, EdgeMutablePropertyGraph*
	adjacency_matrix	VertexListGraph, EdgeListGraph, IncidenceGraph, AdjacencyGraph, AdjacencyMatrix, VertexMutablePropertyGraph, EdgeMutablePropertyGraph
BGL adaptors	edge_list	EdgeListGraph
	std::vector<EdgeList>	VertexListGraph, IncidenceGraph, AdjacencyGraph
custom	[your own]	Whatever you make it support.

+ ...

Example

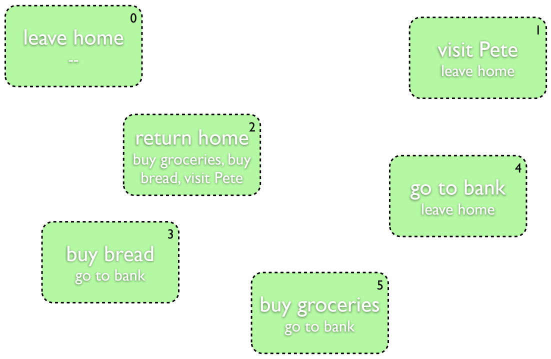
Example
Problem Statement

Scheduling is important in various contexts: scheduling calculations for CPUs, scheduling processes in buisinesses, scheduling engineering tasks in automated manufactoring.

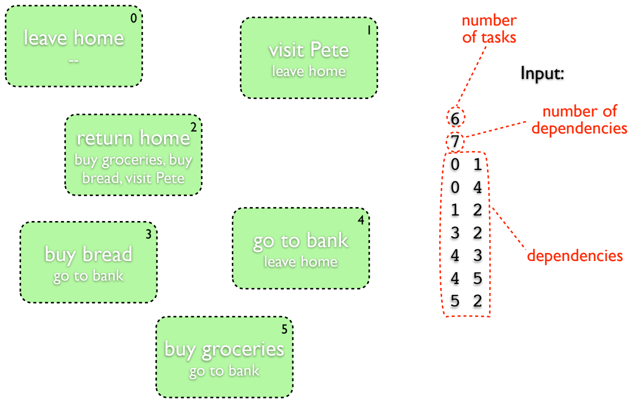
We consider the general case of conflict-free scheduling of abstract tasks. Every task may depend upon the prior execution of other tasks and may be selected only once these tasks have been performed.

Problem: Find an ordering of the tasks such that all dependencies are respected.

Example
Illustrated



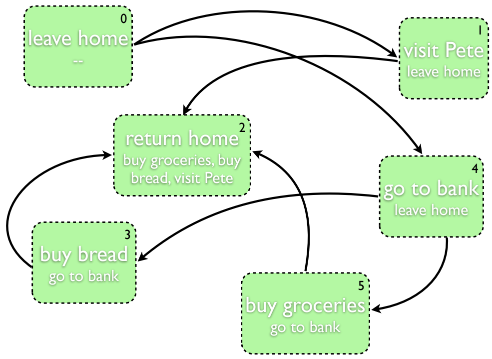
Example
Illustrated



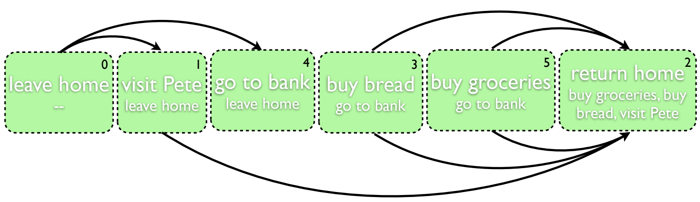
Example
Workflow

1. Decide what algorithm you need
2. Check the documentation for the graph concepts required by the algorithm
3. Choose a suitable graph implementation
4. Set-up the graph and invoke algorithm

Example
Identifying the Underlying Problem



Example
Identifying the Underlying Problem



⇒ Topological Sort

Example
Workflow

1. Decide what algorithm you need
2. Check the documentation for the graph concepts required by the algorithm
3. Choose a suitable graph implementation
4. Set-up the graph and invoke algorithm

Example
Topological Sort Algorithm

topological sort requires: VertexListGraph IncidenceGraph

topological_sort

```
template <typename VertexListGraph, typename OutputIterator,
        typename P, typename T, typename R>
void topological_sort(VertexListGraph& g, OutputIterator result,
    const bgl_named_params<P, T, R>& params = all defaults)
```

The topological sort algorithm creates a linear ordering of the vertices such that if edge *(u,v)* appears in the graph, then *v* comes before *u* in the ordering. The graph must be a directed acyclic graph (DAG). The implementation consists mainly of a call to `depth_first_search()`.

Where Defined:

`boost/graph/topological_sort.hpp`

Parameters

IN: `VertexListGraph& g`

A directed acylic graph (DAG). The graph type must be a model of Vertex List Graph and Incidence Graph. If the graph is not a DAG then a `not_a_dag` exception will be thrown and the user should discard the contents of `result` range.
Python: The parameter is named `graph`.

OUT: `OutputIterator result`

The vertex descriptors of the graph will be output to the `result` output iterator in **reverse** topological order. The iterator type must model Output Iterator.
Python: This parameter is not used in Python. Instead, a Python `list` containing the vertices in topological order is returned.

Example
Workflow

topological sort requires: VertexListGraph IncidenceGraph

1. Decide what algorithm you need
2. Check the documentation for the graph concepts required by the algorithm
3. Choose a suitable graph implementation
4. Set-up the graph and invoke algorithm

Example
Choosing Graph Representation

topological sort requires: VertexListGraph IncidenceGraph

type	Graph	models
BGL classes	adjacency_list	DefaultConstructible, CopyConstructible, VertexListGraph, EdgeListGraph, Inciden AdjacencyGraph, VertexMutableGraph, EdgeMutableGraph, BidirectionalGraph*, VertexMutablePropertyGraph*, EdgeMutablePropertyGraph*
	adjacency_matrix	VertexListGraph, EdgeListGraph, IncidenceGraph, AdjacencyGraph, AdjacencyMatrix, VertexMutablePropertyGraph, EdgeMutablePropertyGraph
BGL adaptors	edge_list	EdgeListGraph
	std::vector<EdgeList>	VertexListGraph, IncidenceGraph, AdjacencyGraph
custom	[your own]	Whatever you make it support.

Easy to use!

Example
Choosing Graph Representation

topological sort requires: VertexListGraph IncidenceGraph

Almost always appropriate!

type	Graph	models
BGL classes	adjacency_list	DefaultConstructible, CopyConstructible, VertexListGraph, EdgeListGraph, Inciden AdjacencyGraph, VertexMutableGraph, EdgeMutableGraph, BidirectionalGraph*, VertexMutablePropertyGraph*, EdgeMutablePropertyGraph*
	adjacency_matrix	VertexListGraph, EdgeListGraph, IncidenceGraph, AdjacencyGraph, AdjacencyMatrix, VertexMutablePropertyGraph, EdgeMutablePropertyGraph
BGL adaptors	edge_list	EdgeListGraph
	std::vector<EdgeList>	VertexListGraph, IncidenceGraph, AdjacencyGraph
custom	[your own]	Whatever you make it support.

Example
Workflow

1. Decide what algorithm you need
2. Check the documentation for the graph concepts required by the algorithm
3. Choose a suitable graph implementation
4. Set-up the graph and invoke algorithm

Example
Graph Setup

```
#include <iostream>
#include <boost/graph/adjacency_list.hpp>
using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, directedS> Graph;

int main()
{
  int n, m; cin >> n >> m;
  Graph graph(n);

  for(int i = 0; i < m; ++i)
  {
    int u, v; cin >> u >> v;
    add_edge(u, v, graph);
  }
}
```

Example
Invoking BGL

```
...
#include <vector>
#include <boost/graph/topological_sort.hpp>
...

int main()
{
  ...

  vector<int> order(n);
  topological_sort(graph, &order[0]);

  for(int i = n-1; i >= 0; --i)
    cout << order[i] << (i > 0 ? " " : "\n");
}
```


Summary

Summary
Workflow

1. Decide what algorithm you need
2. Check the documentation for the graph concepts required by the algorithm
3. Choose a suitable graph implementation
4. Set-up the graph and invoke algorithm

The End.