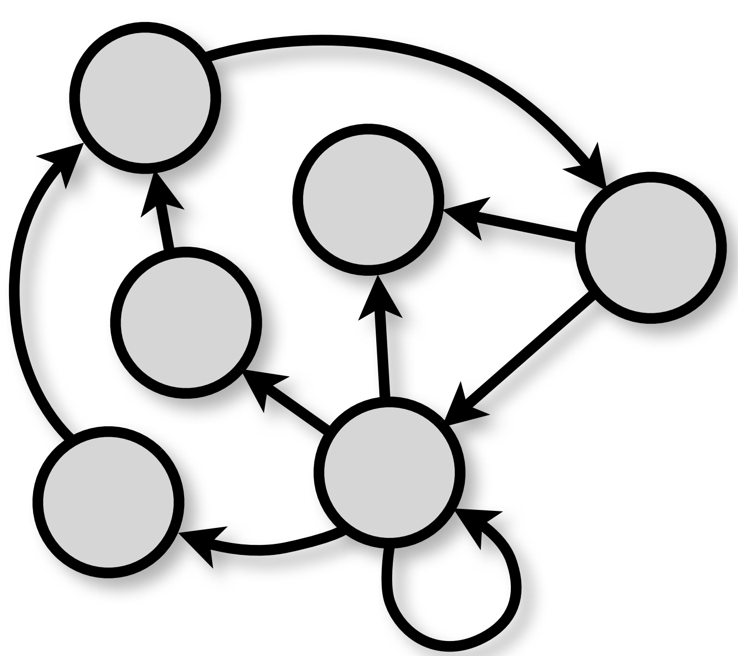
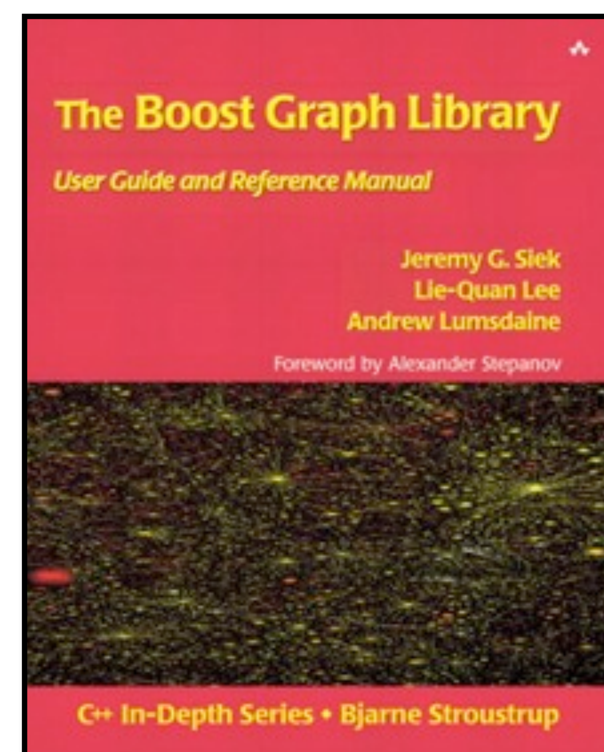
The Boost Graph Library

http://boost.org/libs/graph





Graph Applications

Networks

Compilers

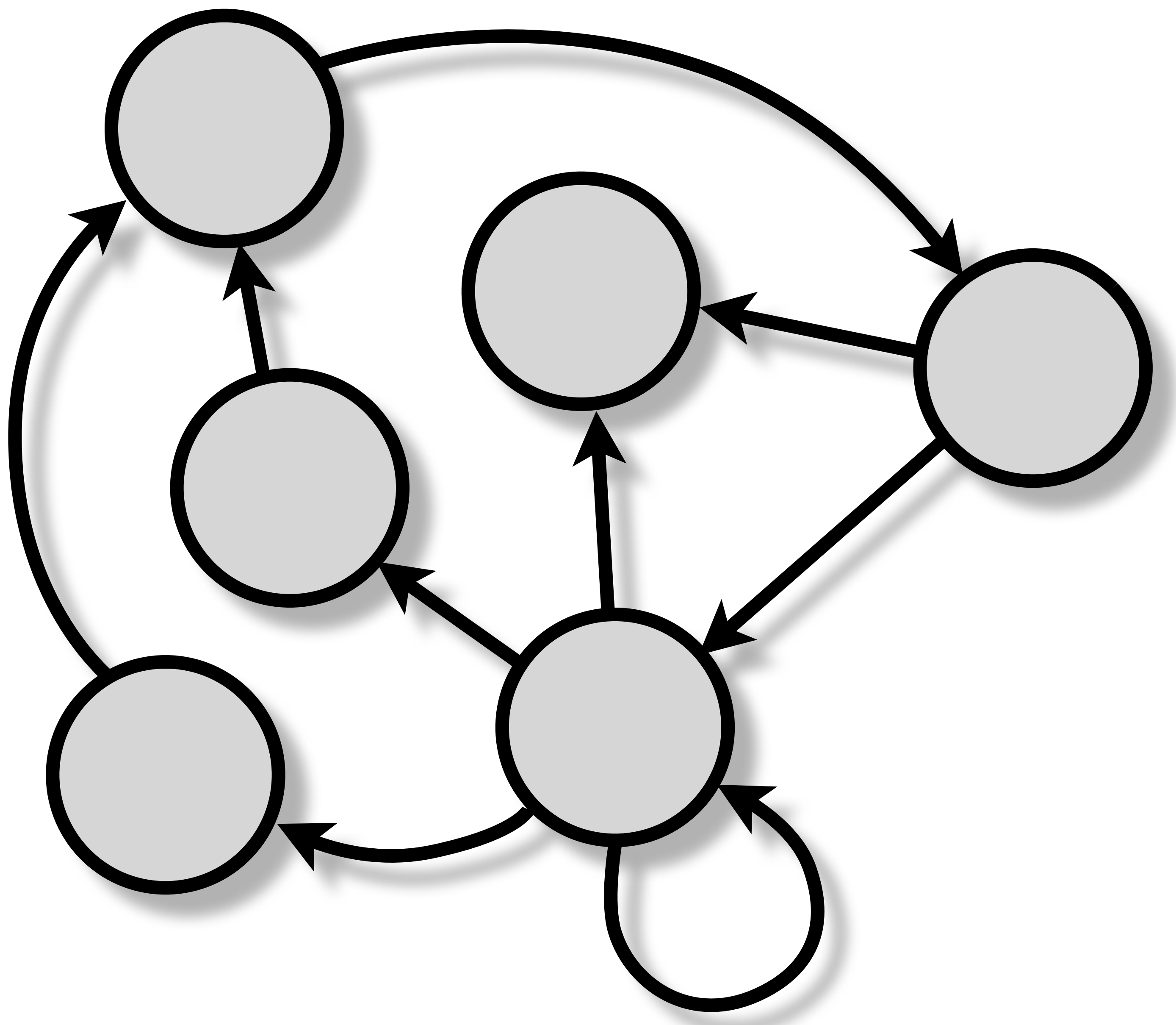
*EDA, GUI, GIS**

Scheduling

Finance

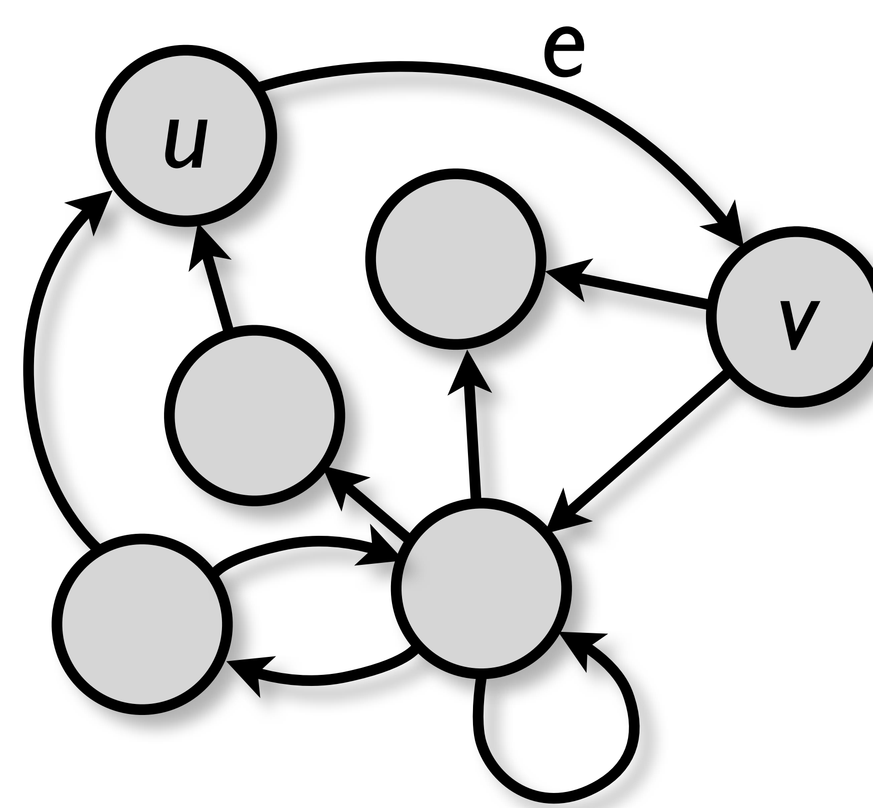
Linear Algebra

Scientific Computing…



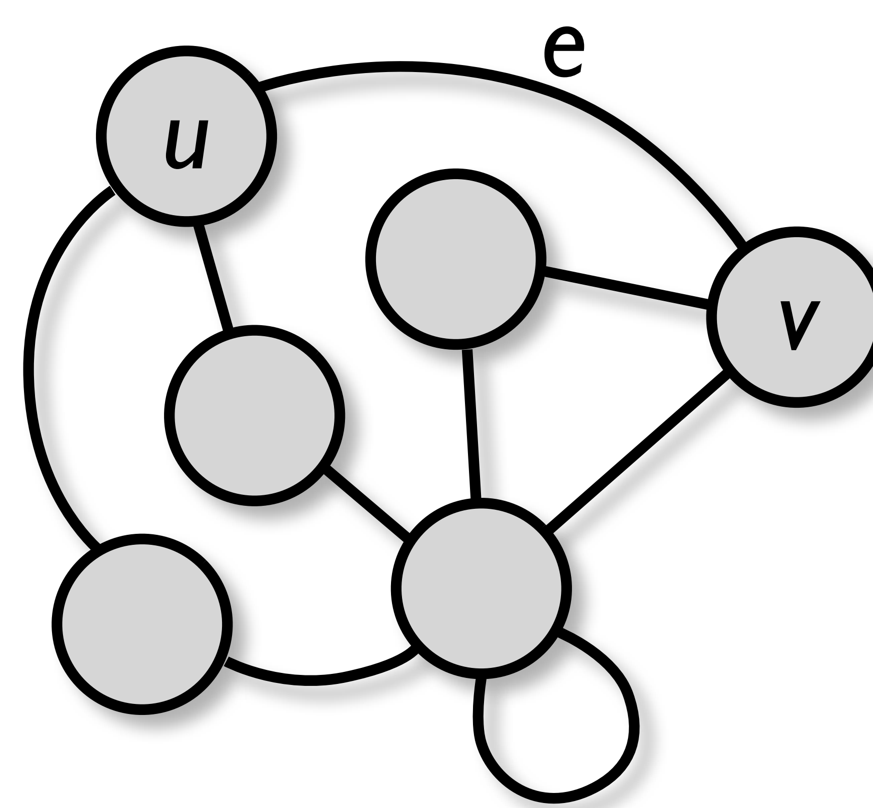
* *and other TLAs*

Graph Abstraction



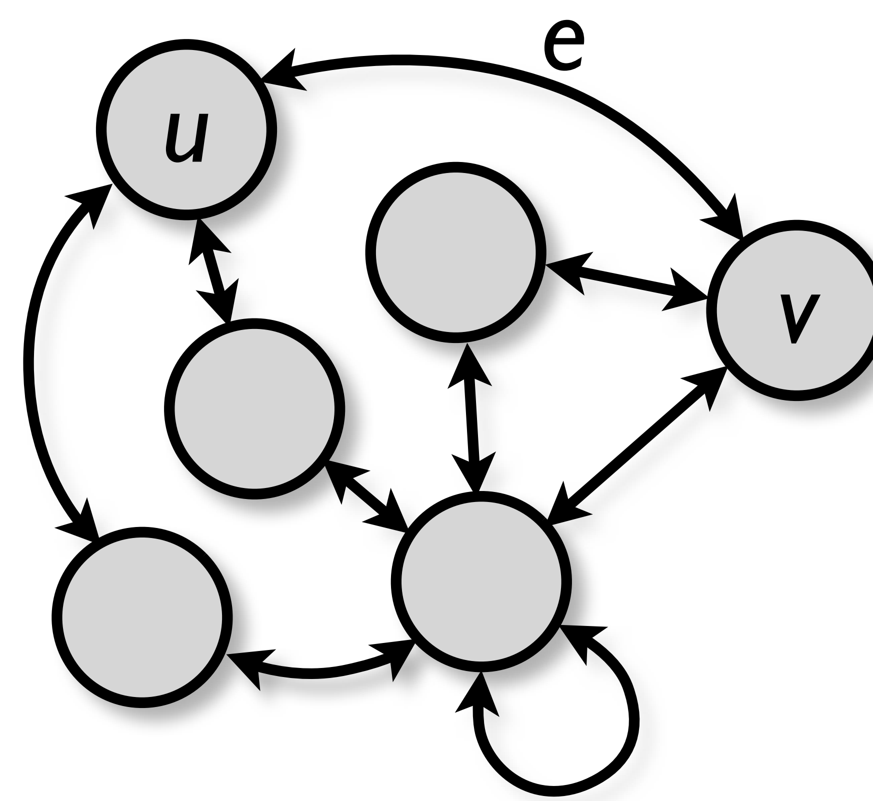
- Vertex set V
- Edge set $E \subseteq V \times V$
- Graph $G = (V, E)$
- Directed Graph:
  - $e \in E$ is an ordered pair (u,v)
  - e is an out-edge of source vertex u
  - e is an in-edge of target vertex v
- Undirected Graph: $(u,v) \equiv (v,u)$

Graph Abstraction



- Vertex set V
- Edge set $E \subseteq V \times V$
- Graph $G = (V, E)$
- Directed Graph:
  - e $\in E$ is an ordered pair (u,v)
  - e is an out-edge of source vertex u
  - e is an in-edge of target vertex v
- Undirected Graph: $(u,v) \equiv (v,u)$

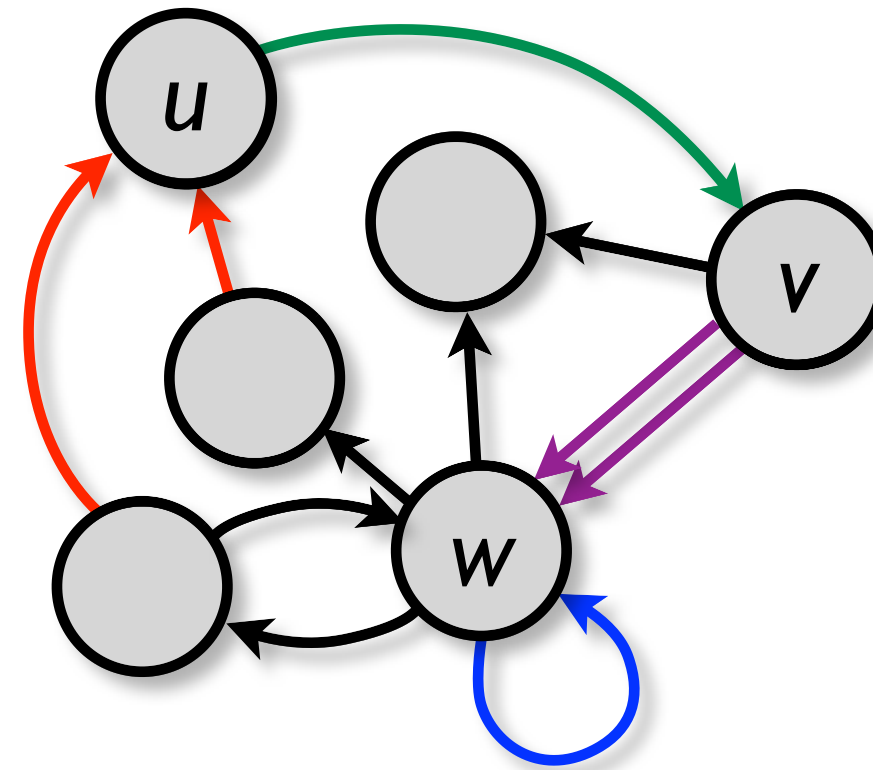
Graph Abstraction



- Vertex set V
- Edge set $E \subseteq V \times V$
- Graph $G = (V, E)$
- Directed Graph:
  - $e \in E$ is an ordered pair (u,v)
  - e is an out-edge of source vertex u
  - e is an in-edge of target vertex v
- Undirected Graph: $(u,v) \equiv (v,u)$

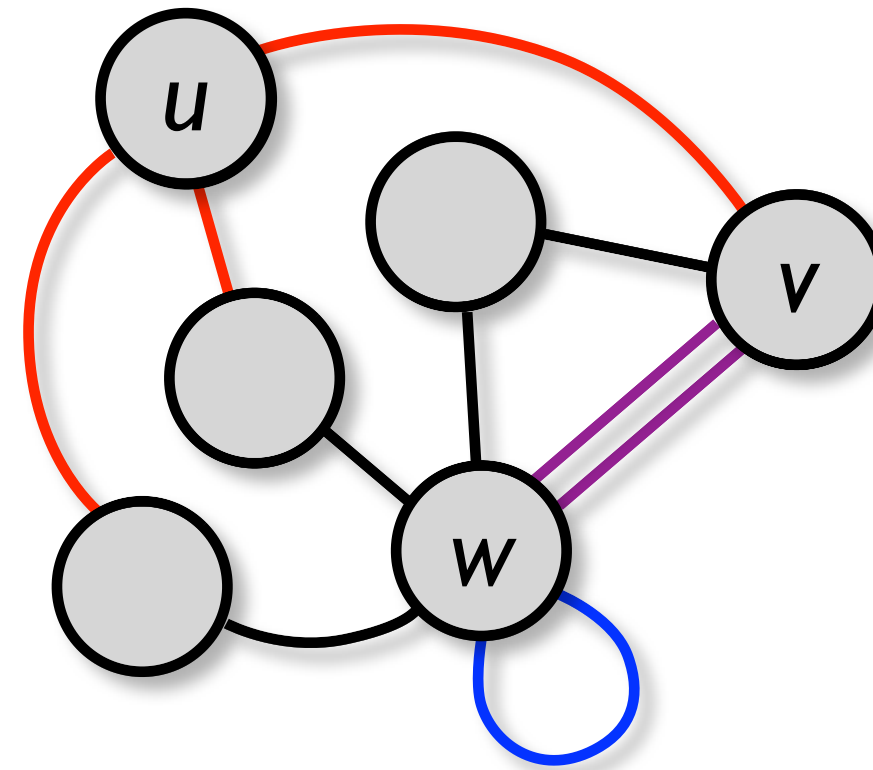
boostpro computing

Directed Graph Terms



- u has in-degree 2
- u has out-degree 1
- $(u,v) \in E \Leftrightarrow v$ is adjacent to u
- (w,w) is a self-loop
- In a multigraph, parallel edges are allowed

Undirected Graph Terms



- *u* has degree 3
- $(u,v) \in E \Leftrightarrow v$ is adjacent to *u*
- (w,w) is a self-loop
- In a multigraph, parallel edges are allowed

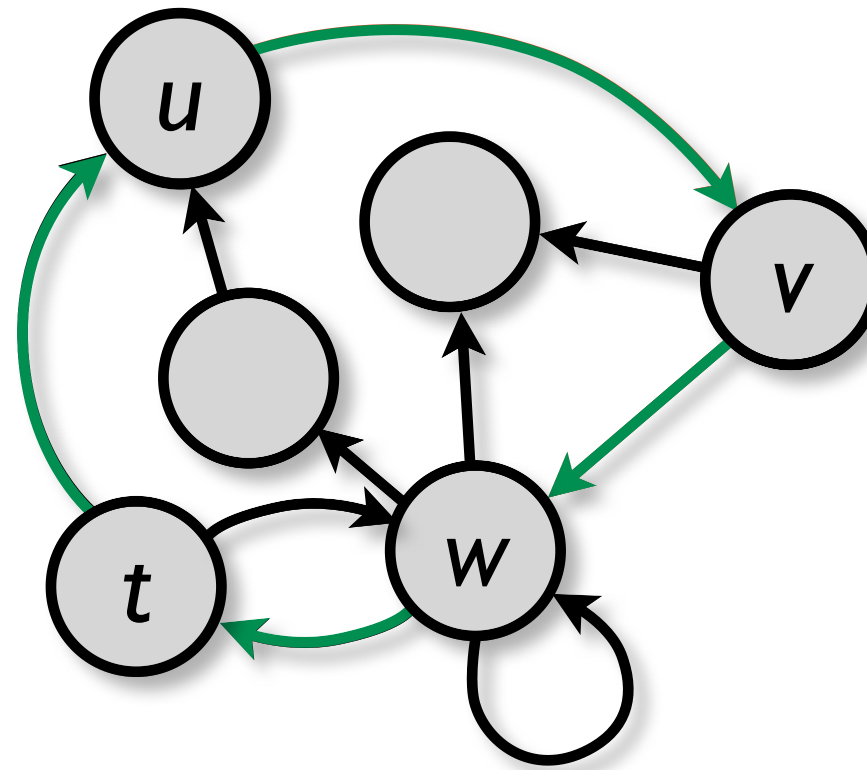
boostpro computing

Paths

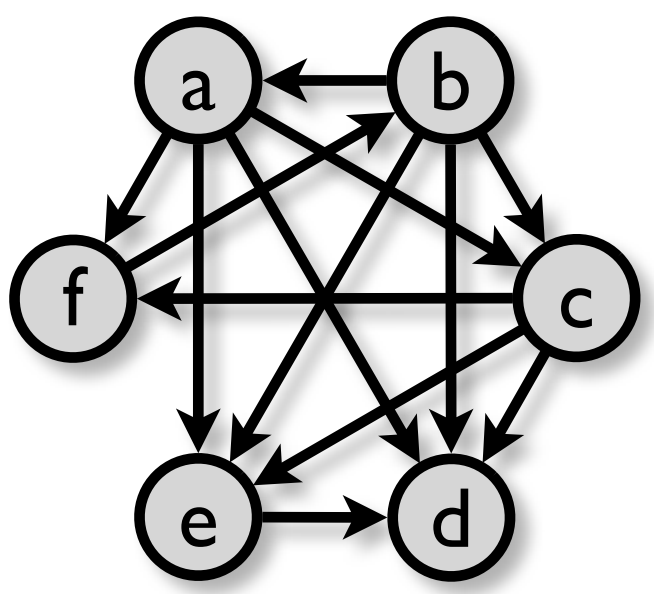
- Path: a sequence of edges such that the target of each edge is the source of its successor, e.g. [(*u*,*v*), (*v*,*w*)]
- *w* is reachable from *u* iff there is a path from *u* to *w*
- Cycle: a path that starts and ends with the same vertex, e.g.

  [(*t*,*u*), (*u*,*v*), (*v*,*w*), (*w*,*t*)]

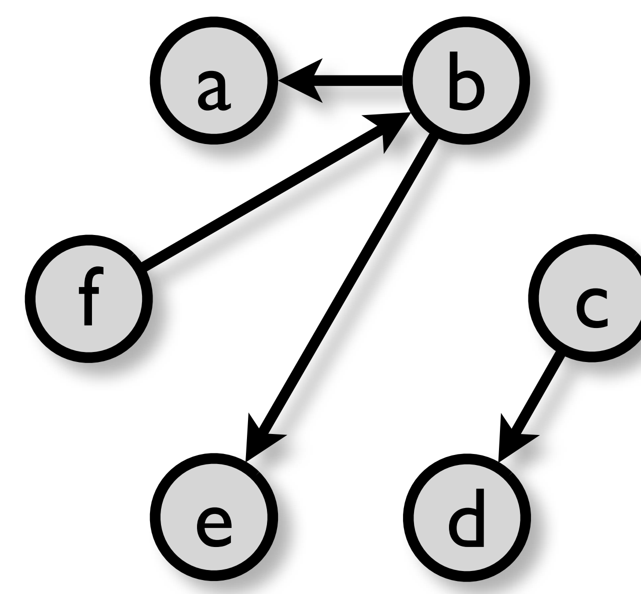
- A graph with no cycles is acyclic



Sparsity $\alpha = \frac{|E|}{|V|}$



<u>Dense</u>: $\alpha \cong |V|$

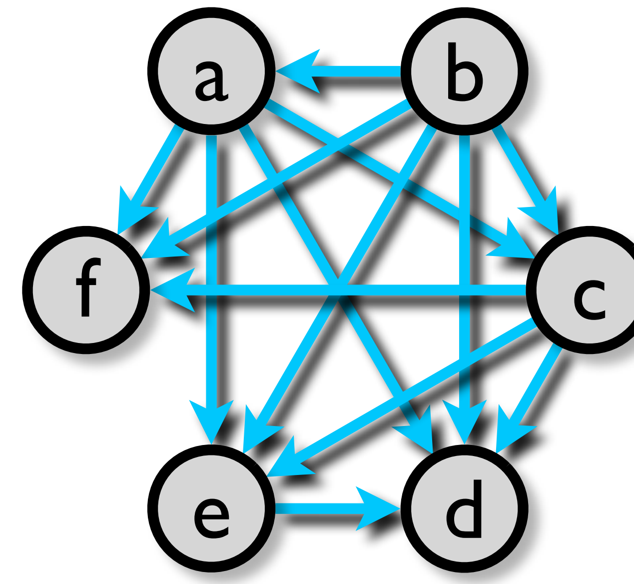


<u>Sparse</u>: $\alpha \ll |V|$

Graph Representations

Adjacency Matrix

- Space: $O(|V|^2)$
- Add edge: $O(1)$
- Query edge: $O(1)$
- Remove edge: $O(1)$
- Add Vertex: $O(|V|^2)$
- Remove Vertex: $O(|V|^2)$

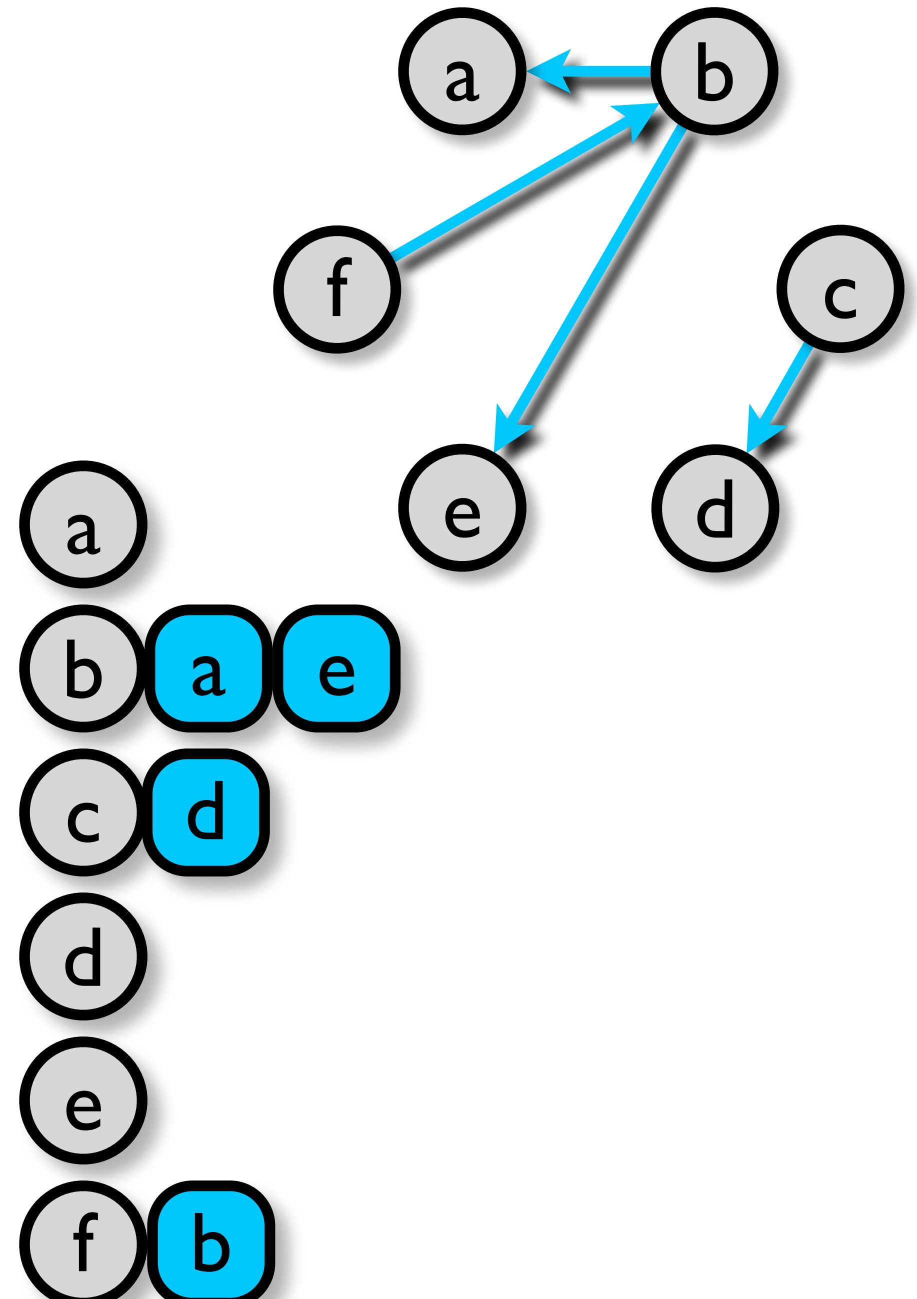


	a	b	c	d	e	f
a	0	0	1	0	1	1
b	1	0	1	1	1	1
c	0	0	0	1	0	1
d	0	0	0	0	0	0
e	0	0	0	1	0	0
f	0	0	0	0	0	0

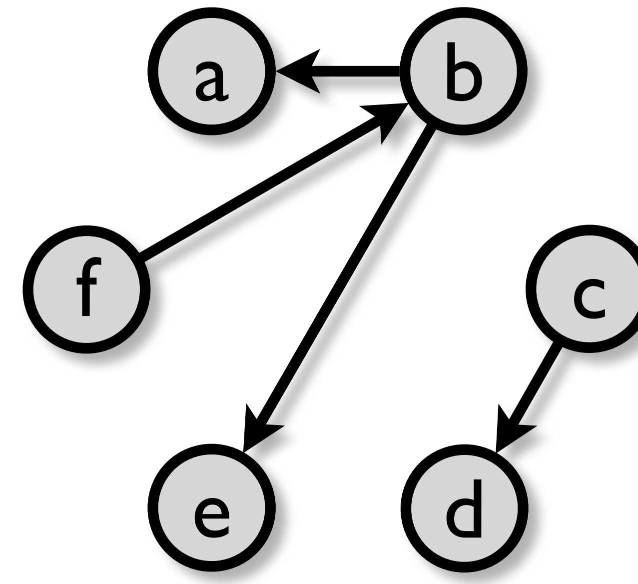
boostpro computing

Adjacency List

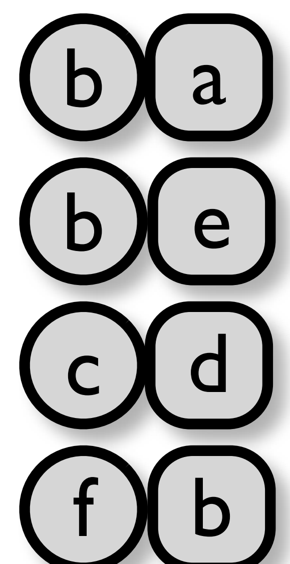
- Space: O(|V|+|E|)
- Add edge: O(1)
- Query edge: O(α)
- Remove edge: O(α)
- Add Vertex: O(1)
- Remove Vertex: O(|V|)



Edge List



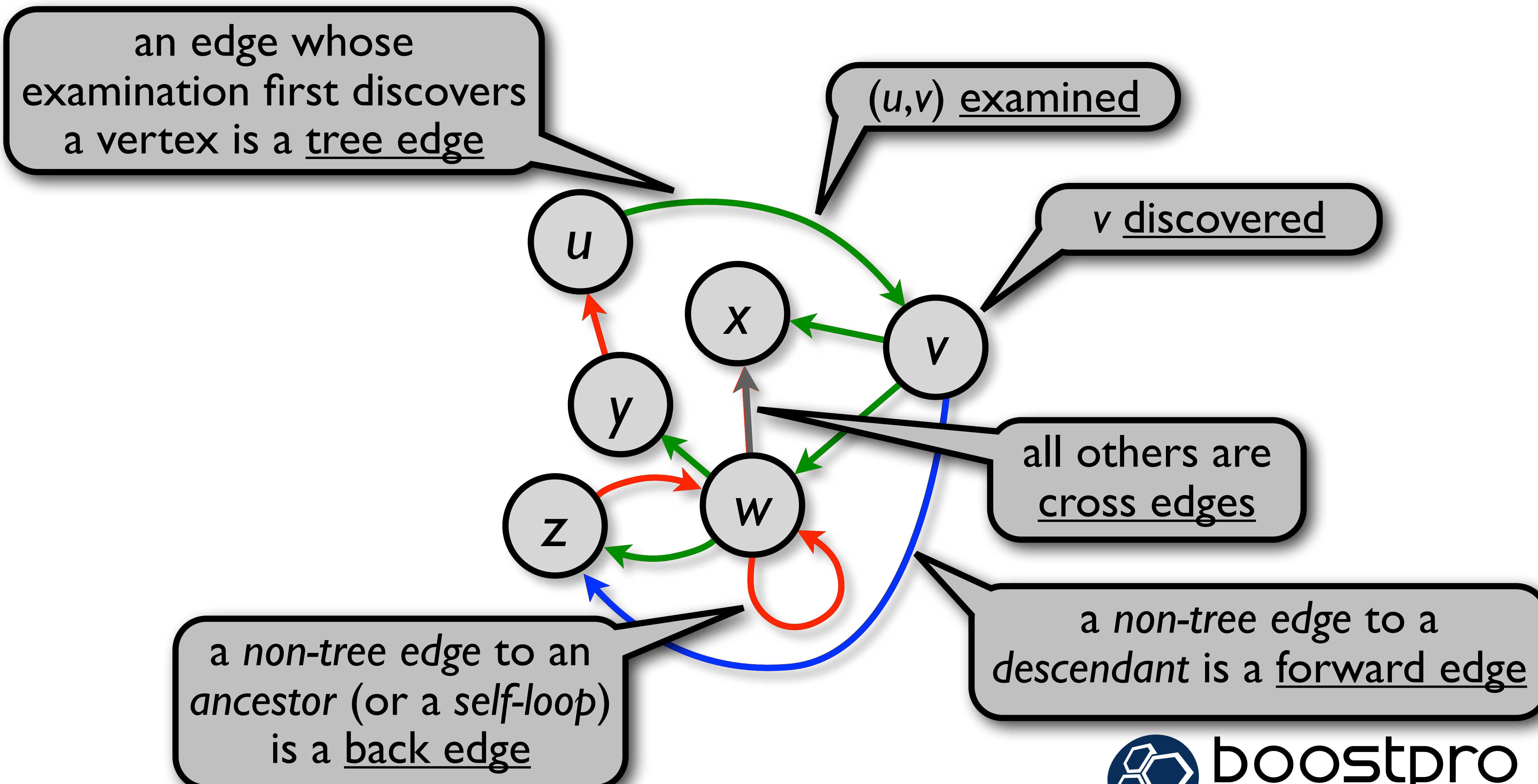
- Space: O(|E|)
- Add edge: O(1)
- Query edge: O(|E|)
- Remove edge: O(|E|)
- Add Vertex: O(?)
- Remove Vertex: O(|E|)



Graph Traversals

Foundational Algorithms

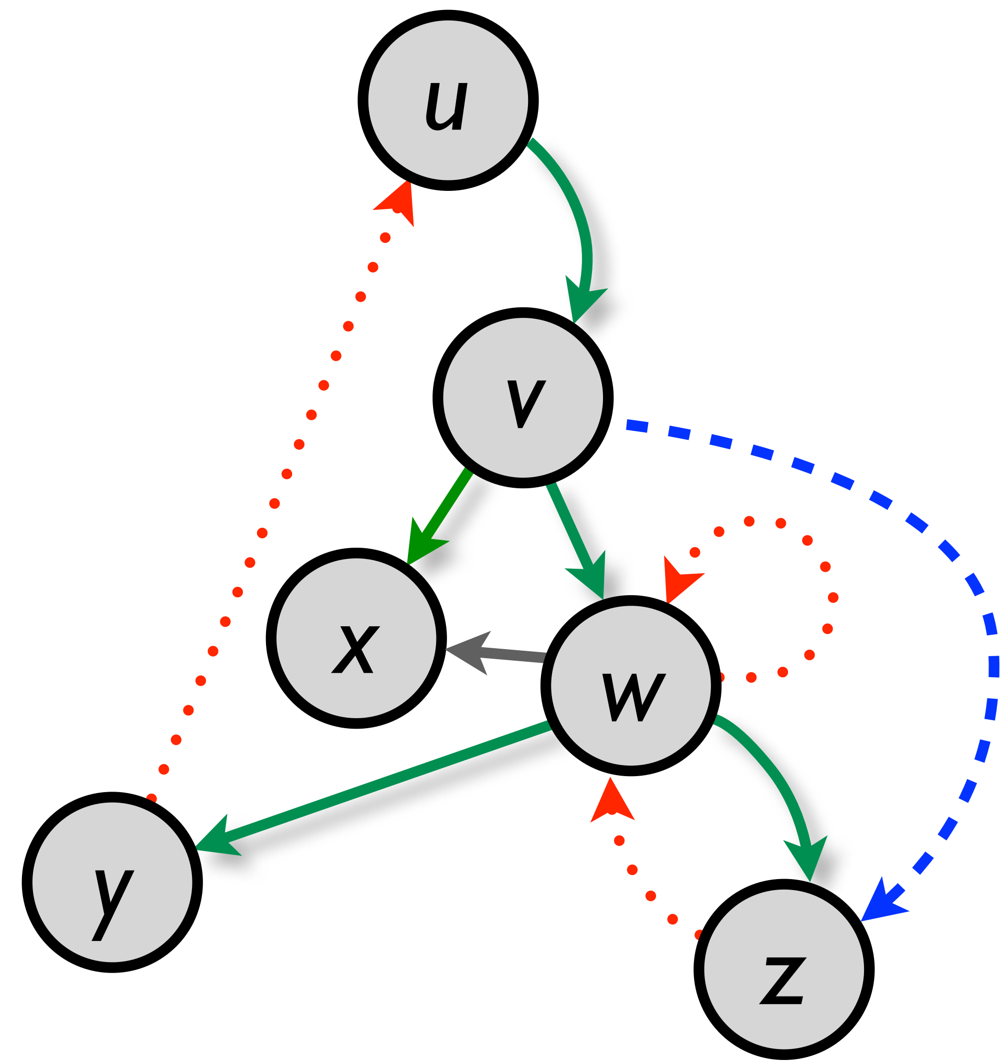
Graph Traversal / Edge Classification



boostpro computing

Graph Traversal / Edge Classification

- Tree edges form one or more trees
- Forward edges point to descendants
- Back edges point to ancestors (and self).
- Cross edges are the others



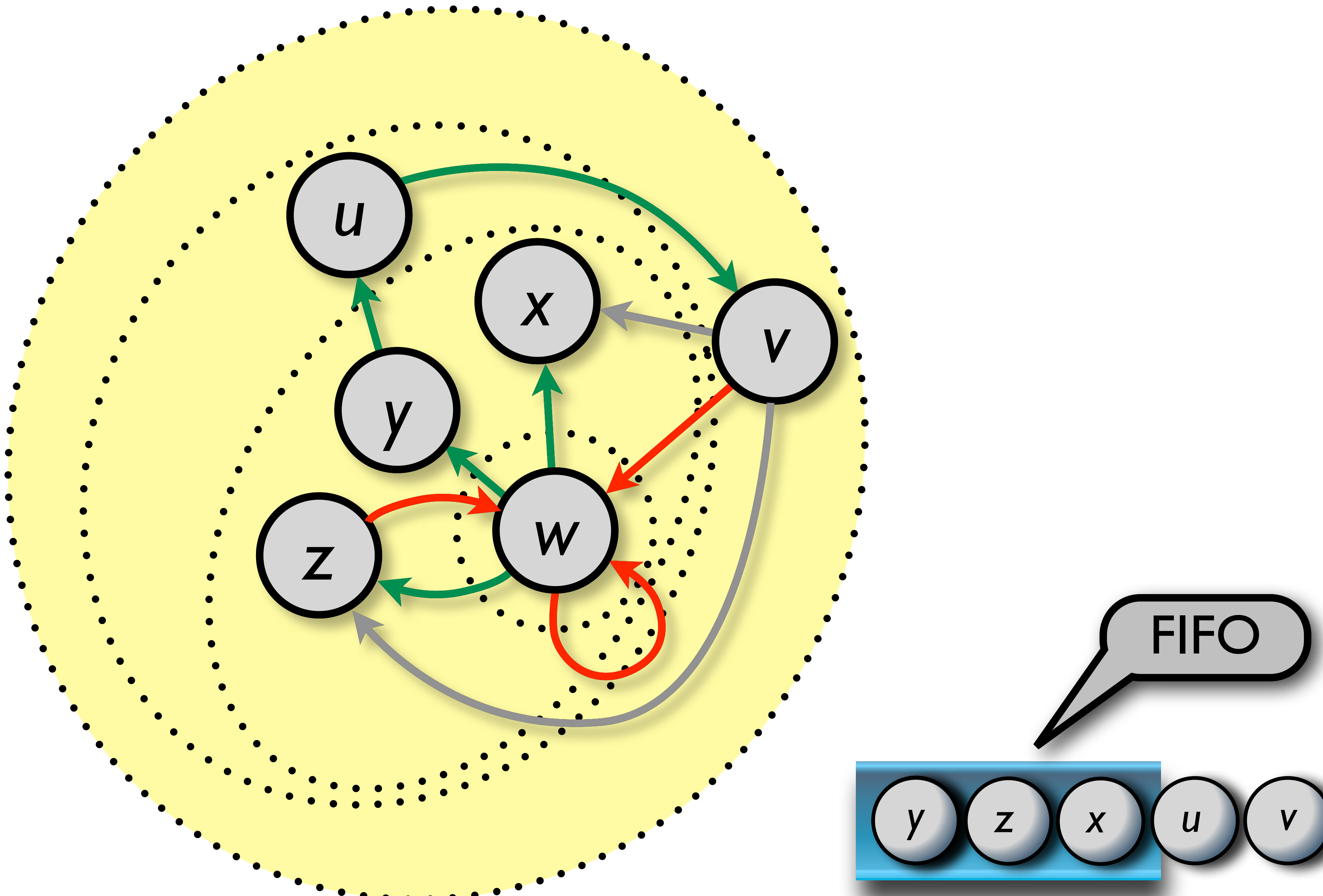
boostpro computing

Breadth-First Search

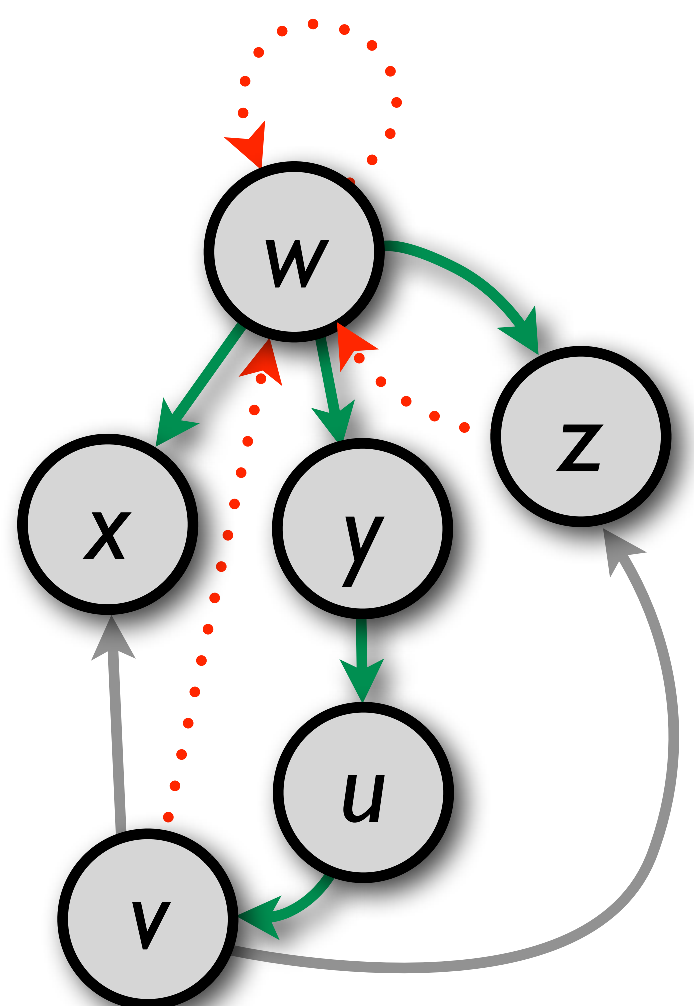
BFS(G, s)
    for each vertex $u \in V[G]$
        $color[u] \leftarrow WHITE$ ◁ initialize vertex u
    $color[s] \leftarrow GRAY$ ◁ discover vertex s
    ENQUEUE(Q, s)
    while ($Q \neq \varnothing$)
        $u \leftarrow$ DEQUEUE(Q) ◁ discover vertex u
        for each $v \in Adj[u]$
            if ($color[v] = WHITE$) ◁ examine edge (u, v)
                $color[v] \leftarrow GRAY$ ◁ (u, v) is a tree edge
                ENQUEUE(Q, v)
            else
                … ◁ (u,v) is a back or cross edge
        $color[u] \leftarrow BLACK$ ◁ finish vertex u

boostpro computing

Breadth-First Traversal



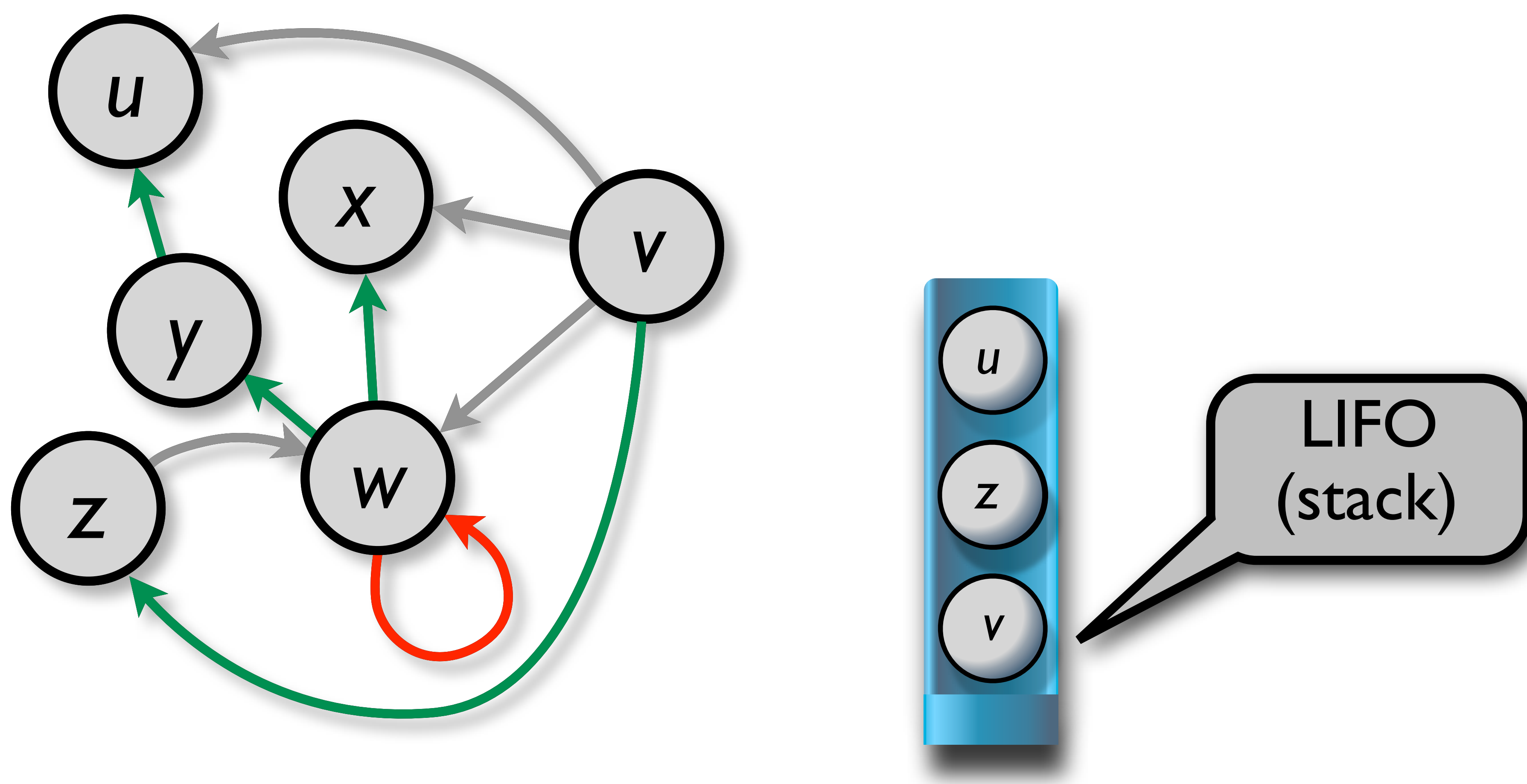
Breadth-First Traversal



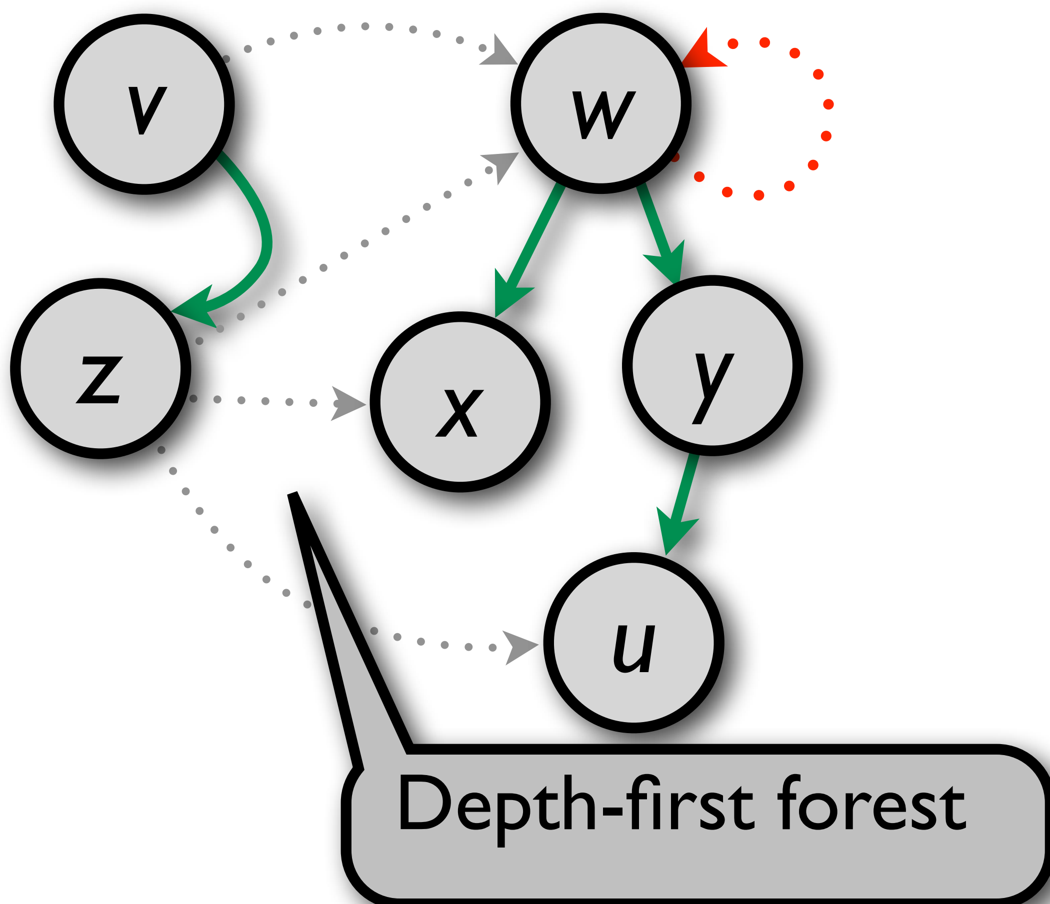
Depth-First Search

```
DFS(G)
  for each vertex u ∈ V[G]
    color[u] ← WHITE                  ◁ initialize vertex u
  for each vertex u ∈ V[G]
    if (color[v] = WHITE)
      call DFS-VISIT(G, u)

DFS-VISIT(G, u)
  color[u] ← GRAY                     ◁ discover vertex u
  for each v ∈ Adj[u]
    if (color[v] = WHITE)             ◁ examine edge (u, v)
      call DFS-VISIT(G, u)            ◁ (u, v) is a tree edge
    else if (color[v] = GRAY)
      …                               ◁ (u,v) is a back edge
    else
      …                               ◁ (u,v) is a cross or forward edge
  color[u] ← BLACK                    ◁ finish vertex u
```

Depth-First Traversal



Depth-First Traversal



Boost Graph Library

Motivation and Design Principles

Earlier Graph Libraries

- One of two categories
  - Easy to use but inefficient (LEDA, GTL)
  - Fast but incomprehensible (Fortran)
- All were inflexible:
  - Tied to their own graph data structures, …
  - …which have have hard-coded *properties*
  - Algorithms not *extensible*

Properties

- Data associated with vertices and edges
- Storage:
  - Internal: in graph data structure
  - External: in other data structure, addressed by vertex/edge id

Purpose	Example(s)
Input	edge length/flow/ capacity
State	vertex/edge color parent vertex in tree
Output	shortest path length/ predecessor
Static	Vertex index, out-degree
User	vertex/edge label, etc.

Algorithm Extension

```
BFS(G, s)
  for each vertex u ∈ V[G]
    c[u] ← WHITE
  c[s] ← GRAY
  ENQUEUE(Q, s)
  while (Q ≠ ∅)
    u ← DEQUEUE(Q)
    for each v ∈ Adj[u]
      if (c[v] = WHITE)
        c[v] ← GRAY
        ENQUEUE(Q, v)
      else
        …
    c[u] ← BLACK
```

```
DIJKSTRA(G, s)
  for each vertex u ∈ V[G]
    d[u] ← ∞
  d[s] ← 0
  ENQUEUE(Q, s)
  while (Q ≠ ∅)
    u ← DEQUEUE(Q)
    for each v ∈ Adj[u]
      if (d[v] > weight(u,v)+d[u])
        d[v] ← weight(u,v)+d[u]
        ENQUEUE-RELAX(Q, v)
      else
        …
```

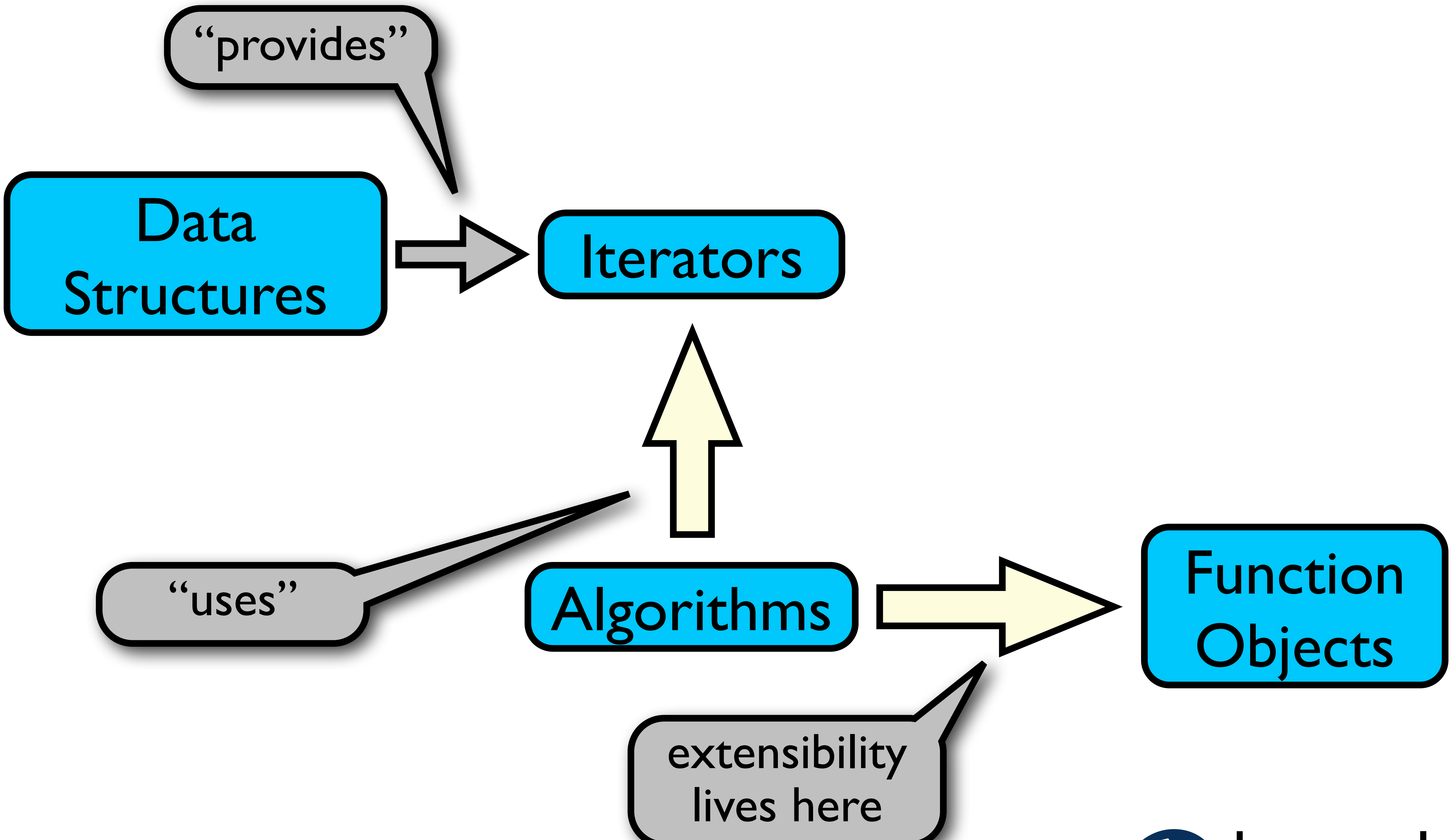
Desiderata: Axes of Genericity

- Graph representation category
- Details of specific graph representation
- Choice of vertex and edge properties
- Details of vertex and edge property storage
- Algorithm/data-structure decoupling
- Algorithm composability
- Algorithm extensibility

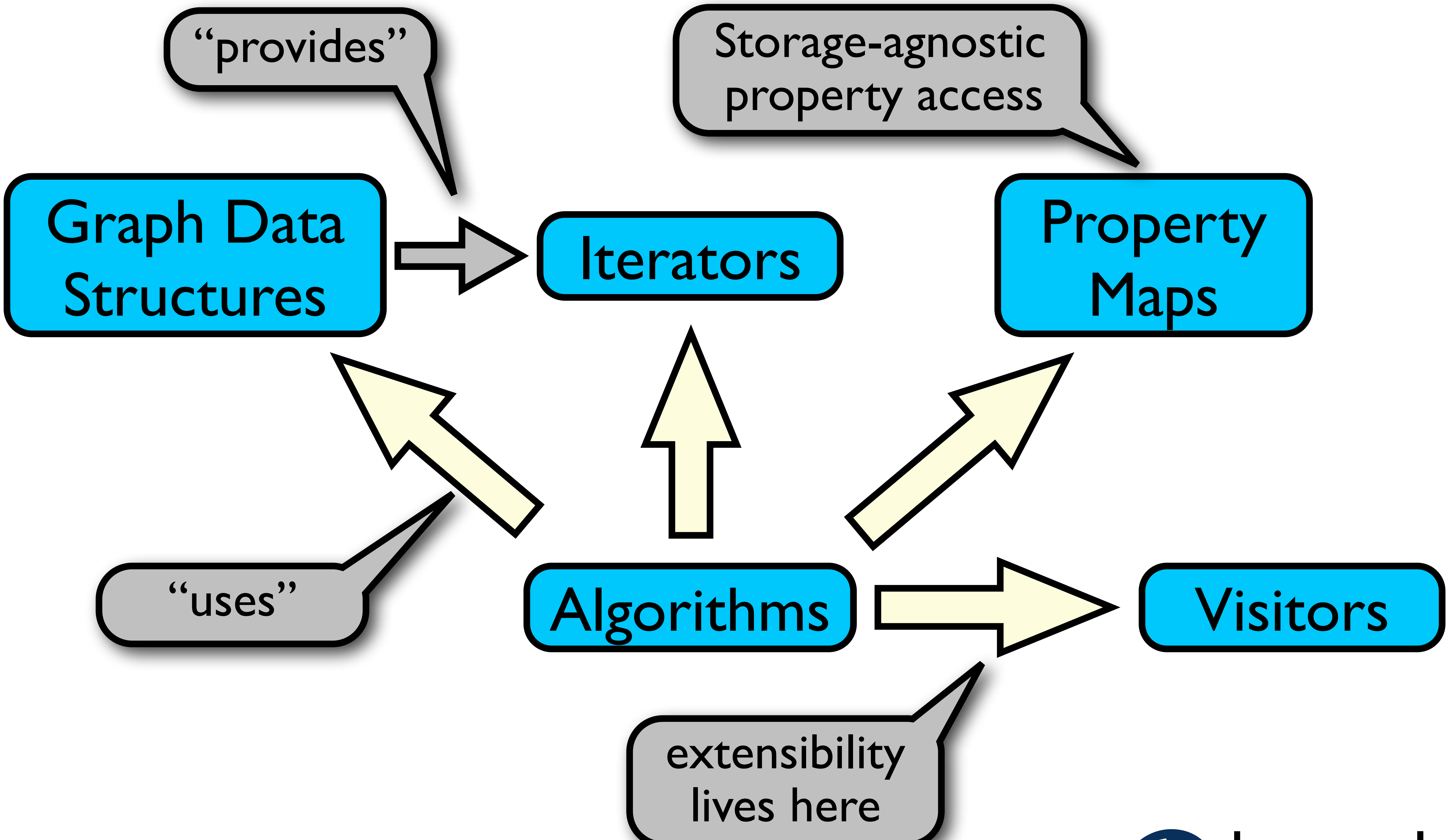
Desiderata: General

- High Performance
- Natural
  - Similar to pseudocode in literature
  - Close to existing libraries
  - Compatible with STL and built-in types

STL Architecture



BGL Architecture



BGL Algorithms Cheatsheet

- **Basic Operations**
  - copy_graph
  - transpose_graph
- **Core Searches**
  - breadth_first_search
  - breadth_first_visit
  - depth_first_search
  - depth_first_visit
  - undirected_dfs
- **Other Core**
  - topological_sort
  - transitive_closure
  - lengauer_tarjan_dominator_tree
- **Shortest Paths/Cost Minimization**
  - dijkstra_shortest_paths
  - dijkstra_shortest_paths_no_color_map
  - bellman_ford_shortest_paths
  - dag_shortest_paths
  - johnson_all_pairs_shortest_paths
  - floyd_warshall_all_pairs_shortest_paths
  - resource-constrained shortest paths
  - astar_search
- **Minimum Spanning Tree**
  - kruskal_minimum_spanning_tree
  - prim_minimum_spanning_tree
- **Connected Components**
  - connected_components
  - strong_components
  - biconnected_components
  - articulation_points
  - <u>Incremental Connected Components</u>
    - initialize_incremental_components
    - incremental_components
    - same_component
    - component_index
- **Maximum Flow and Matching**
  - edmonds_karp_max_flow
  - push_relabel_max_flow
  - kolmogorov_max_flow
  - edmonds_maximum_cardinality_matching
- **Sparse Matrix Ordering**
  - cuthill_mckee_ordering
  - king_ordering
  - minimum_degree_ordering
  - sloan_ordering
  - sloan_start_end_vertices
- **Graph Metrics**
  - ith/max/aver/rms_wavefront
  - bandwidth
  - ith_bandwidth
  - brandes_betweenness_centrality
  - minimum/maximum_cycle_ratio
- **Structure Comparisons**
  - isomorphism
  - mcgregor_common_subgraphs
- **Layout**
  - random_graph_layout
  - circle_layout
  - kamada_kawai_spring_layout
  - fruchterman_reingold_force_directed_layout
  - gursoy_atun_layout
- **Clustering**
  - betweenness_centrality_clustering
- **Planarity**
  - boyer_myrvold_planarity_test
  - planar_face_traversal
  - planar_canonical_ordering
  - chrobak_payne_straight_line_drawing
  - is_straight_line_drawing
  - is_kuratowski_subgraph
  - make_connected
  - make_biconnected_planar
  - make_maximal_planar
- **Miscellaneous**
  - metric_tsp_approx
  - sequential_vertex_coloring

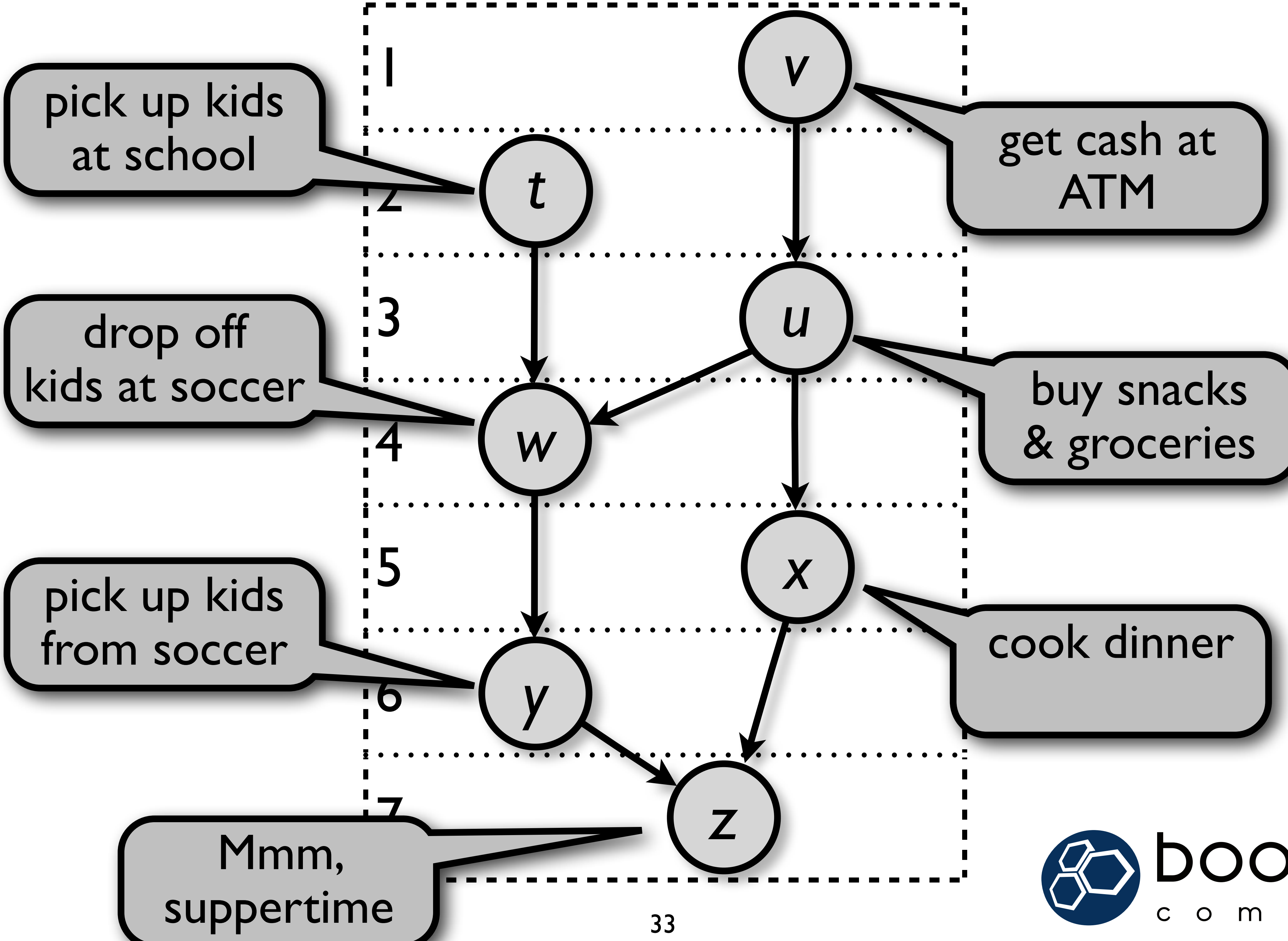
boostpro computing

Hello, BGL!

Example: topological sort

Topological Sort:

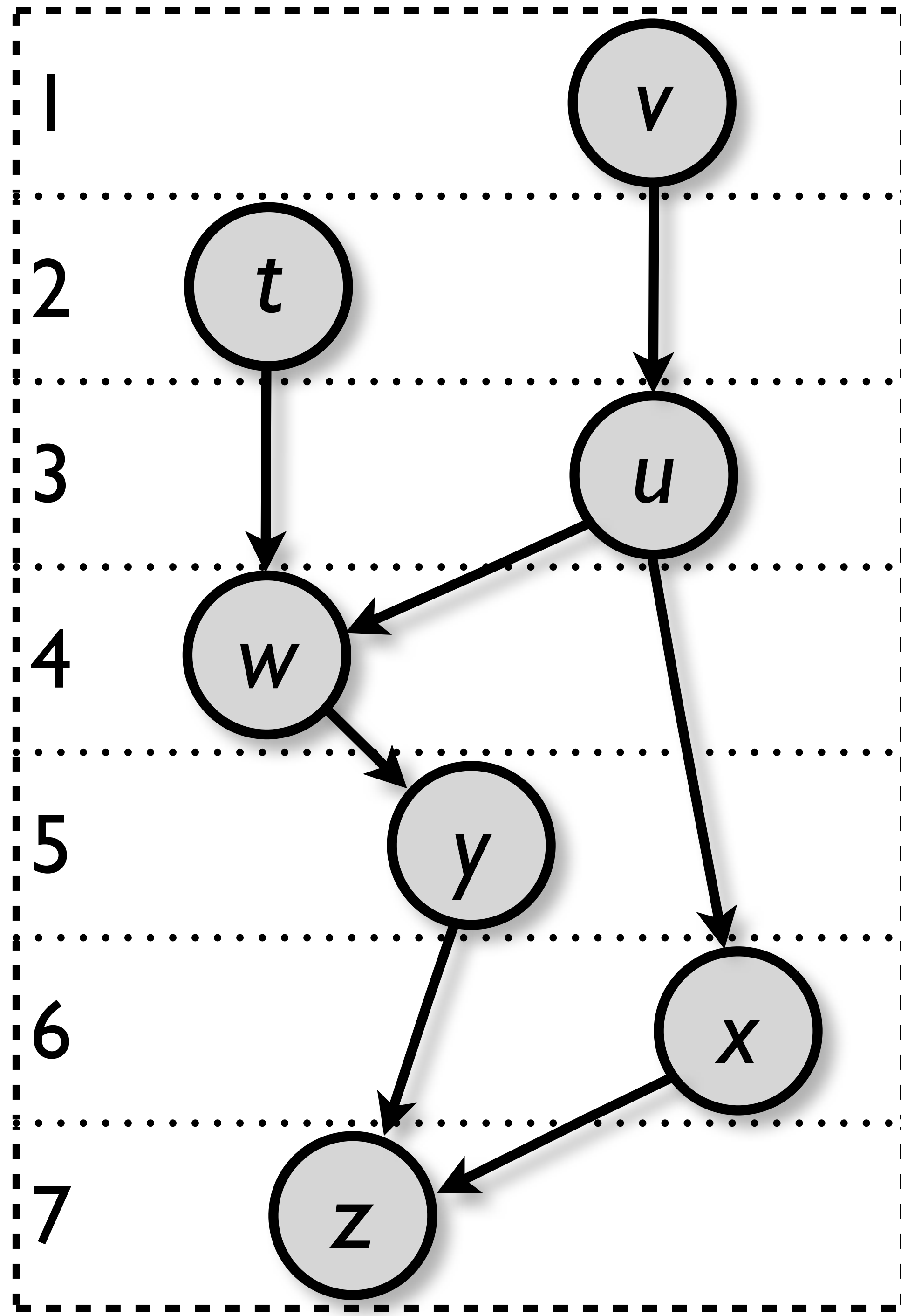
$(u,v) \in E \Longrightarrow u$ precedes v



boostpro computing

Topological Sort:

$(u,v) \in E \Longrightarrow u$ precedes v

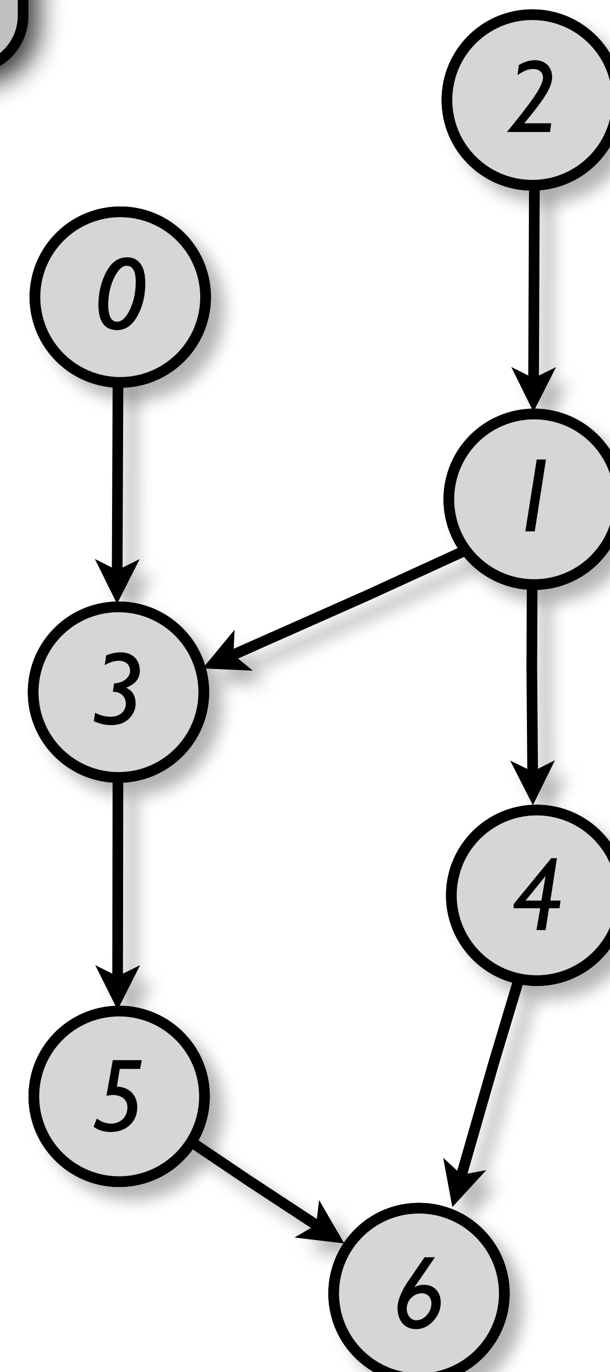


boostpro computing

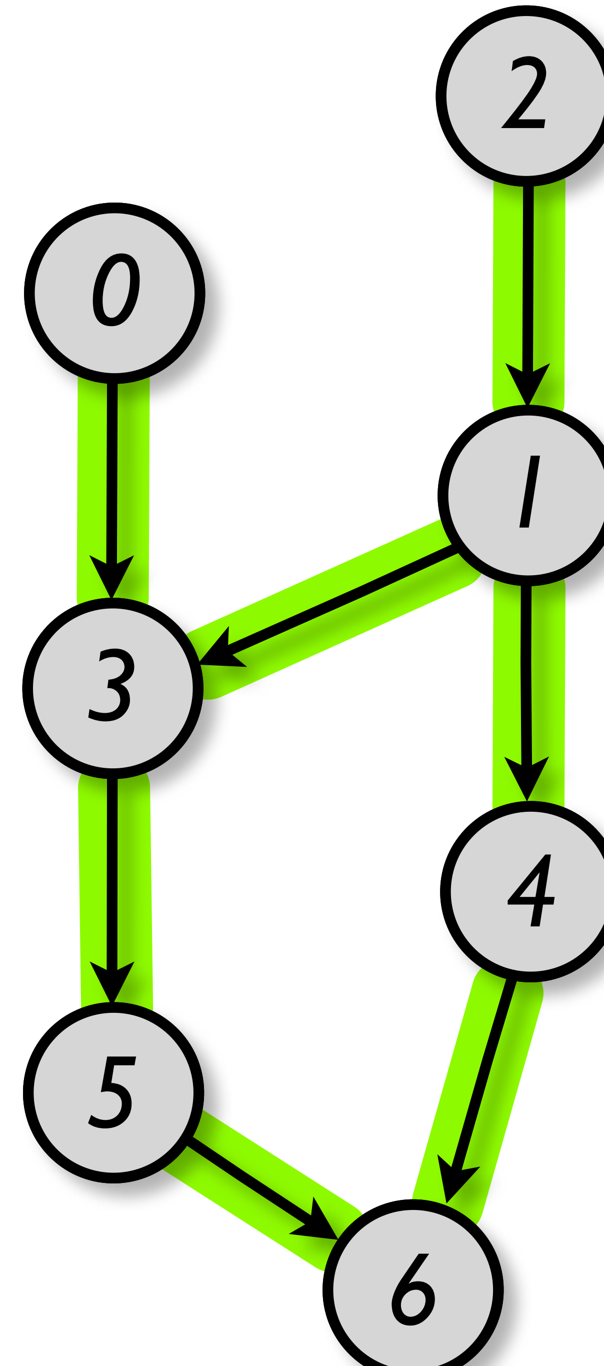
```
            .
            .
            .

// Write g's vertex ids to result_iter in reverse topological order
template <typename Graph, typename OutputIterator>
void topological_sort(Graph& g, OutputIterator result_iter);

            .
            .
            .
```

post-hoc adaptation

```
#include <deque>
#include <vector>
#include <list>
#include <iostream>
#include <boost/graph/vector_as_graph.hpp>
#include <boost/graph/topological_sort.hpp>

const char* tasks[] = {
    "pick up kids from school",           // 0
    "buy groceries (and snacks)",         // 1
    "get cash at ATM",                    // 2
    "drop off kids at soccer practice",   // 3
    "cook dinner",                        // 4
    "pick up kids from soccer",           // 5
    "eat dinner"                          // 6
};
int const n_tasks = sizeof(tasks) / sizeof(char*);

int main()
{
    std::vector< std::list<int> > g(n_tasks);
    g[0].push_back(3);
    g[1].push_back(3);
    g[1].push_back(4);
    g[2].push_back(1);
```

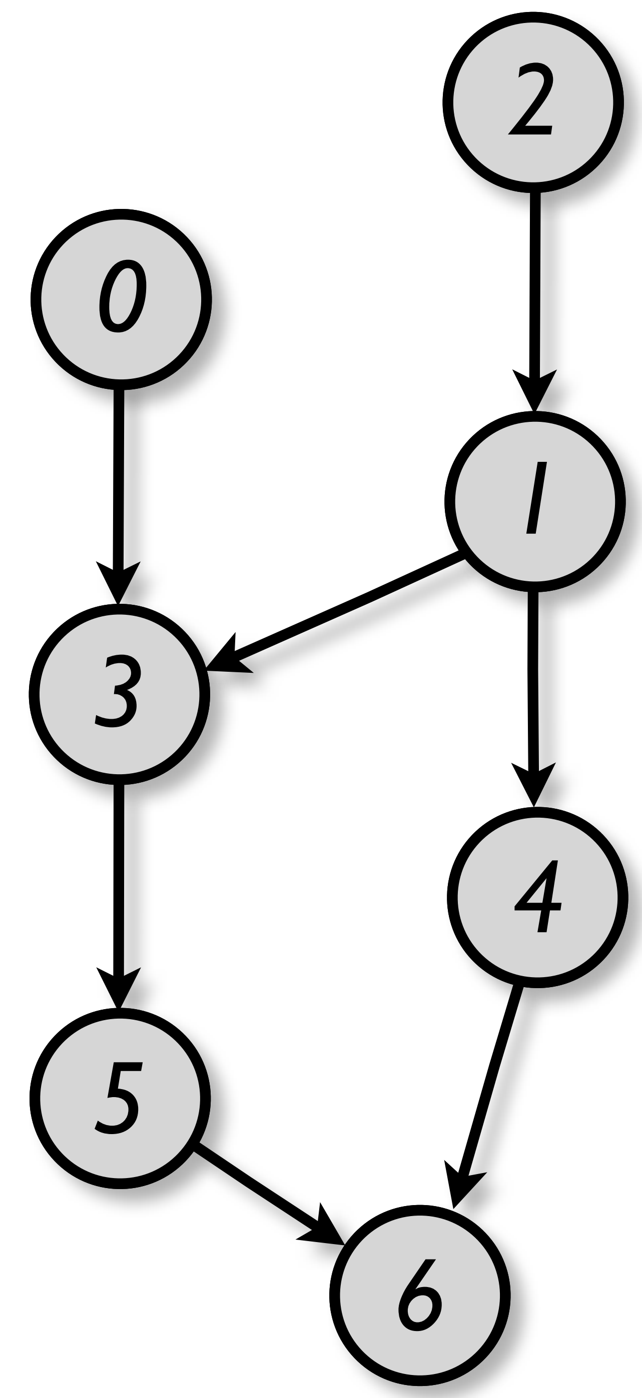


```
int const n_tasks = sizeof(tasks) / sizeof(char*);

int main()
{
    std::vector< std::list<int> > g(n_tasks);
    g[0].push_back(3);
    g[1].push_back(3);
    g[1].push_back(4);
    g[2].push_back(1);
    g[3].push_back(5);
    g[4].push_back(6);
    g[5].push_back(6);

    std::deque<int> topo_order;

    boost::topological_sort(
       g, std::front_inserter(topo_order),
       vertex_index_map(boost::identity_property_map()));

    for (std::deque<int>::iterator i = topo_order.begin();
          i != topo_order.end(); ++i)
    {
       std::cout << tasks[*i] << std::endl;
    }
}
```

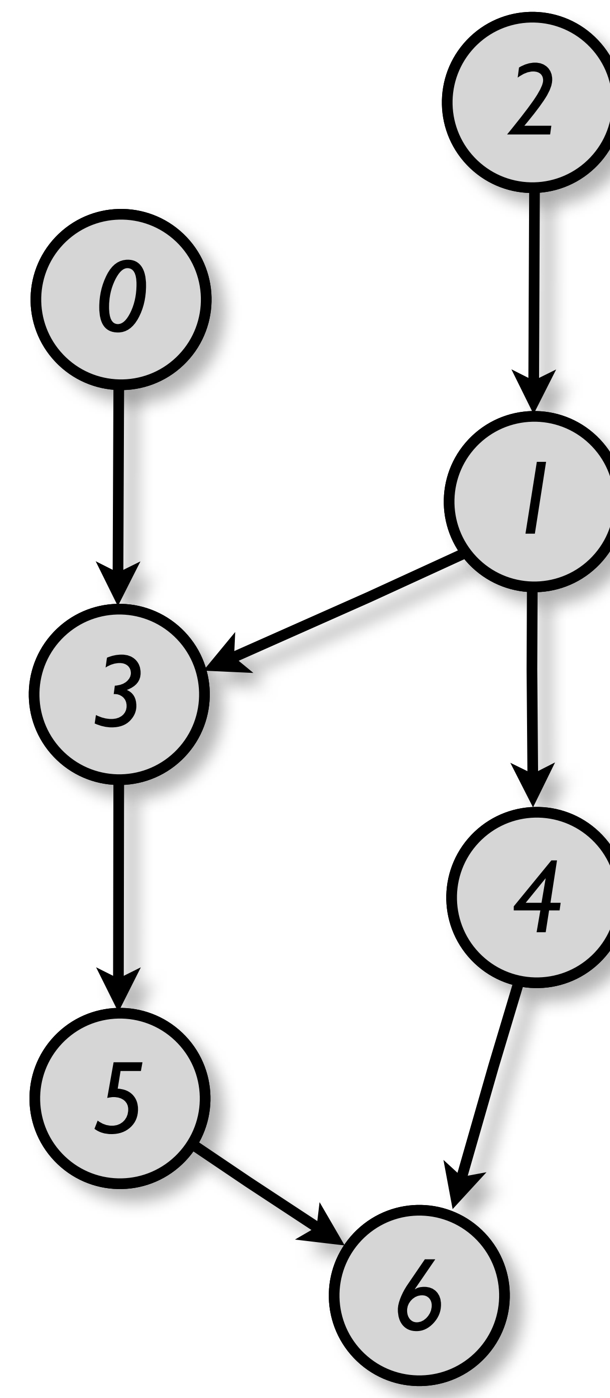


```
$ ./x
get cash at ATM
buy groceries (and snacks)
cook dinner
pick up kids from school
drop off kids at soccer practice
pick up kids from soccer
eat dinner
```

```
int const n_tasks = sizeof(tasks) / sizeof(char*);

int main()
{
    std::vector< std::list<int> > g(n_tasks);
    g[0].push_back(3);
    g[1].push_back(3);
    g[1].push_back(4);
    g[2].push_back(1);
    g[3].push_back(5);
    g[4].push_back(6);
    g[5].push_back(6);

    std::deque<int> topo_order;

    boost::topological_sort(
       g, std::front_inserter(topo_order),
       vertex_index_map(boost::identity_property_map()));

    for (std::deque<int>::iterator i = topo_order.begin();
         i != topo_order.end(); ++i)
    {
       std::cout << tasks[*i] << std::endl;
    }
```



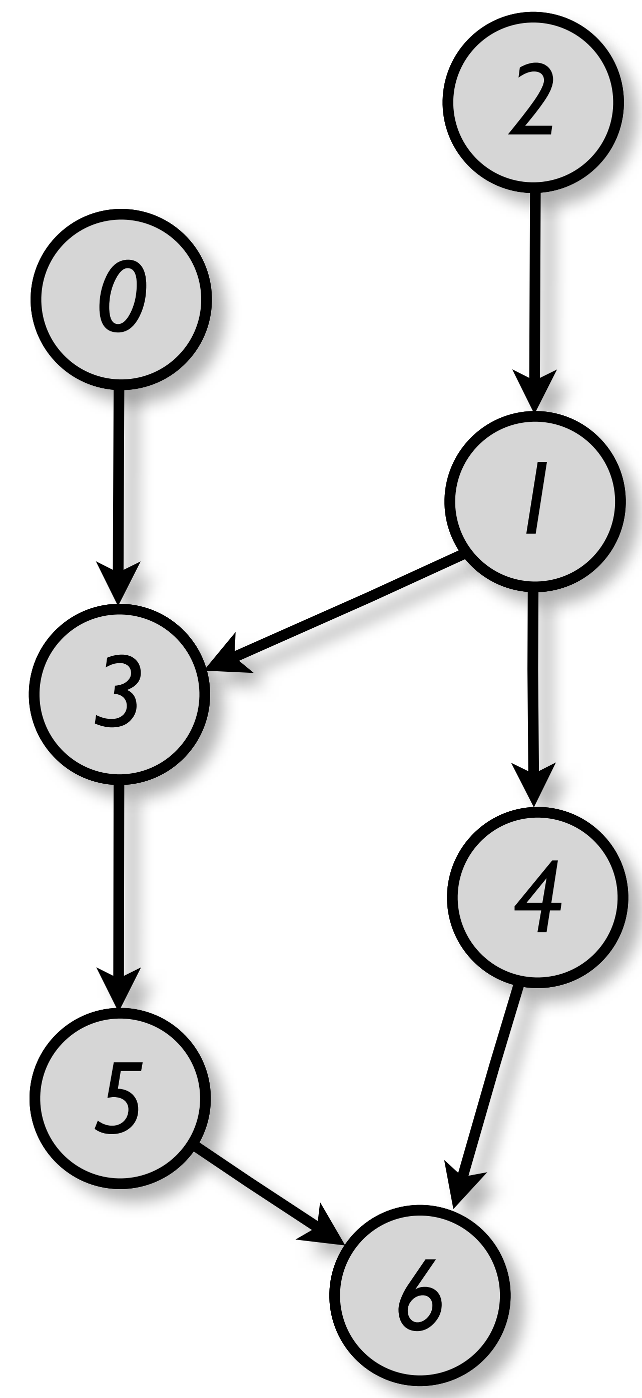
```cpp
#include <boost/graph/vector_as_graph.hpp>
#include <boost/graph/topological_sort.hpp>

const char* tasks[] = {
    "pick up kids from school",           // 0
    "buy groceries (and snacks)",         // 1
    "get cash at ATM",                    // 2
    "drop off kids at soccer practice",   // 3
    "cook dinner",                        // 4
    "pick up kids from soccer",           // 5
    "eat dinner"                          // 6
};
int const n_tasks = sizeof(tasks) / sizeof(char*);

int main()
{
    std::vector< std::list<int> > g(n_tasks);
    g[0].push_back(3);
    g[1].push_back(3);
    g[1].push_back(4);
    g[2].push_back(1);
    g[3].push_back(5);
    g[4].push_back(6);
    g[5].push_back(6);
```



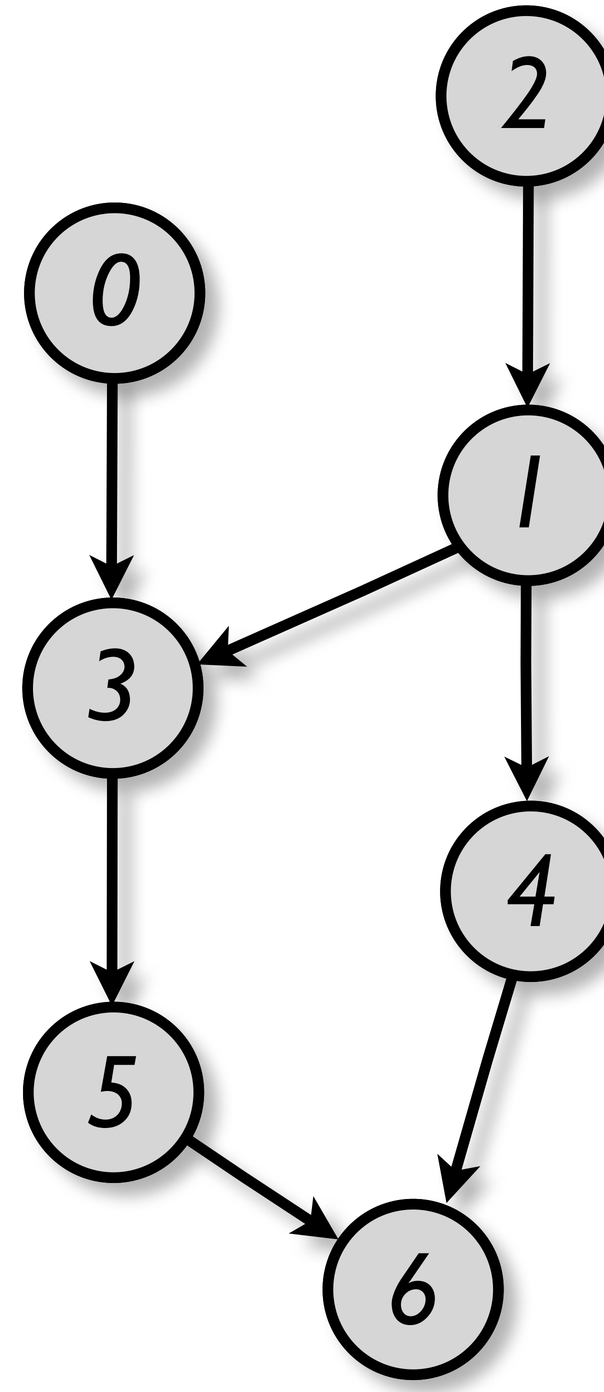
library | client | session

boostpro computing

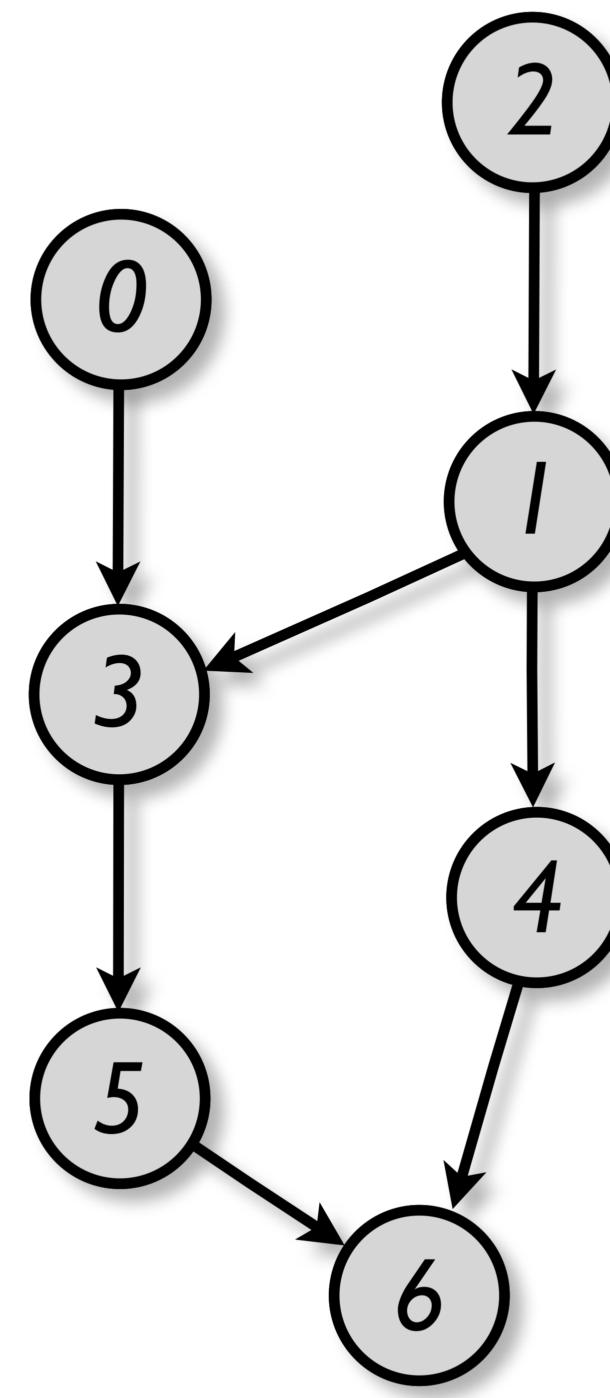

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/topological_sort.hpp>

const char* tasks[] = {
    "pick up kids from school",             // 0
    "buy groceries (and snacks)",           // 1
    "get cash at ATM",                      // 2
    "drop off kids at soccer practice",     // 3
    "cook dinner",                          // 4
    "pick up kids from soccer",             // 5
    "eat dinner"                            // 6
};
int const n_tasks = sizeof(tasks) / sizeof(char*);

int main()
{
    std::vector< std::list<int> > g(n_tasks);
    g[0].push_back(3);
    g[1].push_back(3);
    g[1].push_back(4);
    g[2].push_back(1);
    g[3].push_back(5);
    g[4].push_back(6);
    g[5].push_back(6);
```



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

const char* tasks[] = {
    "pick up kids from school",            // 0
    "buy groceries (and snacks)",          // 1
    "get cash at ATM",                     // 2
    "drop off kids at soccer practice",    // 3
    "cook dinner",                         // 4
    "pick up kids from soccer",            // 5
    "eat dinner"                           // 6
};
int const n_tasks = sizeof(tasks) / sizeof(char*);

int main()
{
    std::vector< std::list<int> > g(n_tasks);
    g[0].push_back(3);
    g[1].push_back(3);
    g[1].push_back(4);
    g[2].push_back(1);
    g[3].push_back(5);
    g[4].push_back(6);
    g[5].push_back(6);
```



```cpp
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/topological_sort.hpp>

const char* tasks[] = {
    "pick up kids from school",             // 0
    "buy groceries (and snacks)",           // 1
    "get cash at ATM",                      // 2
    "drop off kids at soccer practice",     // 3
    "cook dinner",                          // 4
    "pick up kids from soccer",             // 5
    "eat dinner"                            // 6
};
int const n_tasks = sizeof(tasks) / sizeof(char*);

int main()
{
    using namespace boost;
    adjacency_list<listS, vecS, directedS> g(n_tasks);
    add_edge(0, 3, g);
    add_edge(1, 3, g);
    add_edge(1, 4, g);
    add_edge(2, 1, g);
    add_edge(3, 5, g);
    add_edge(4, 6, g);
    add_edge(5, 6, g);
```



```
$ ./x
get cash at ATM
buy groceries (and snacks)
cook dinner
pick up kids from school
drop off kids at soccer practice
pick up kids from soccer
eat dinner
```

This Works, Too

(with appropriate adaptation)

```
typedef struct vertex_struct
{
    struct arc_struct* arcs;
} Vertex;

typedef struct arc_struct
{
    Vertex* tip;
    struct arc_struct* next;
} Arc;

typedef struct graph_struct
{
    Vertex* vertices;
} Graph;
```

Desiderata: Axes of Genericity

- Graph representation category
- ✓ Details of specific graph representation
- Choice of vertex and edge properties
- Details of vertex and edge property storage
- ✓ Algorithm/data-structure decoupling
- Algorithm composability
- Algorithm extensibility

Boost Graph Library

Concepts

Write this Section!

Boost Graph Library

Descriptors, Property Maps, and Visitors

Descriptors

- Meaning and specific representation of "vertex" and "edge" are below the level of graph abstraction.
- BGL must model only ***relationships***.
- <u>Descriptor</u>: a token or ID used to identify a vertex or edge
  - Expected to be a lightweight value type
  - Otherwise opaque from BGL's point-of-view
  - Access types as `graph_traits<G>::vertex_descriptor`, `graph_traits<G>::edge_descriptor`

boostpro computing

Property Maps

- Expected to be a lightweight value type
- Interface: `get(m, key)` `put(m, key, v)`
- Lvalue property maps only: `pmap[key]`
- BGL uses vertex/edge descriptors as keys
- Exterior property map: *passed to algorithm* as argument
- Interior property map: *extracted by algorithm* from graph if no exterior map supplied

boostpro computing

Visitors

- A "multi-callback" bundle
- Interface: `vis.`*eventname*`(` *descriptor*`,` *graph* `)`
- Expected to be a lightweight value type
- Used to extend the functionality of algorithms

Breadth-First Visitor

BFS(G, s)	
for each vertex $u \in V[G]$	
$color[u] \leftarrow WHITE$	◁ `vis.`<u>`initialize_vertex`</u>`(u,G)`
$color[s] \leftarrow GRAY$	◁ `vis.`<u>`discover_vertex`</u>`(u,G)`
ENQUEUE(Q, s)	
while ($Q \neq \emptyset$)	
$u \leftarrow$ DEQUEUE(Q)	◁ `vis.examine_vertex(u,G)`
for each $v \in Adj[u]$	
if ($color[v] = WHITE$)	◁ `vis.`<u>`examine_edge`</u>`((u,v),G)`
$color[v] \leftarrow GRAY$	
	◁ `vis.`<u>`tree_edge`</u>`((u,v),G)`
ENQUEUE(Q, v)	◁ `vis.`<u>`discover_vertex`</u>`(v,G)`
else	
	◁ `vis.non_tree_edge(v,G)`
if ($color[v] = GRAY$)	
	◁ `vis.gray_target((u,v),G)`
else	
	◁ `vis.black_target((u,v),G)`
$color[u] \leftarrow BLACK$	◁ `vis.`<u>`finish_vertex`</u>`(u,G)`

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

const char* tasks[] = {
    "pick up kids from school",           // 0
    "buy groceries (and snacks)",         // 1
    "get cash at ATM",                    // 2
    "drop off kids at soccer practice",   // 3
    "cook dinner",                        // 4
    "pick up kids from soccer",           // 5
    "eat dinner"                          // 6
};
int const n_tasks = sizeof(tasks) / sizeof(char*);

int main()
{
    using namespace boost;
    adjacency_list<listS, vecS, directedS> g(n_tasks);
    add_edge(0, 3, g);
    add_edge(1, 3, g);
    add_edge(1, 4, g);
    add_edge(2, 1, g);
    add_edge(3, 5, g);
    add_edge(4, 6, g);
    add_edge(5, 6, g);
```

```cpp
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/breadth_first_search.hpp>

const char* tasks[] = {
    "pick up kids from school",              // 0
    "buy groceries (and snacks)",            // 1
    "get cash at ATM",                       // 2
    "drop off kids at soccer practice",      // 3
    "cook dinner",                           // 4
    "pick up kids from soccer",              // 5
    "eat dinner"                             // 6
};
int const n_tasks = sizeof(tasks) / sizeof(char*);

int main()
{
    using namespace boost;
    adjacency_list<listS, vecS, directedS> g(n_tasks);
    add_edge(0, 3, g);
    add_edge(1, 3, g);
    add_edge(1, 4, g);
    add_edge(2, 1, g);
    add_edge(3, 5, g);
    add_edge(4, 6, g);
    add_edge(5, 6, g);
```

```
};
int const n_tasks = sizeof(tasks) / sizeof(char*);

struct bfs_counter
  : boost::default_bfs_visitor // inherit empty event point actions
{
    bfs_counter() : ecounter(0), vcounter(0) {}

    template <class Edge, class Graph>
    void tree_edge(Edge e, const Graph&)
    {
       std::cout << "Found tree edge #" << ecounter++ << "\n";
    }

    template <class Vertex, class Graph>
    void discover_vertex(Vertex u, const Graph&)
    {
       std::cout << "Found vertex #" << vcounter++ << "\n";
    }
private:
    std::size_t ecounter, vcounter;
};

int main()
{
```

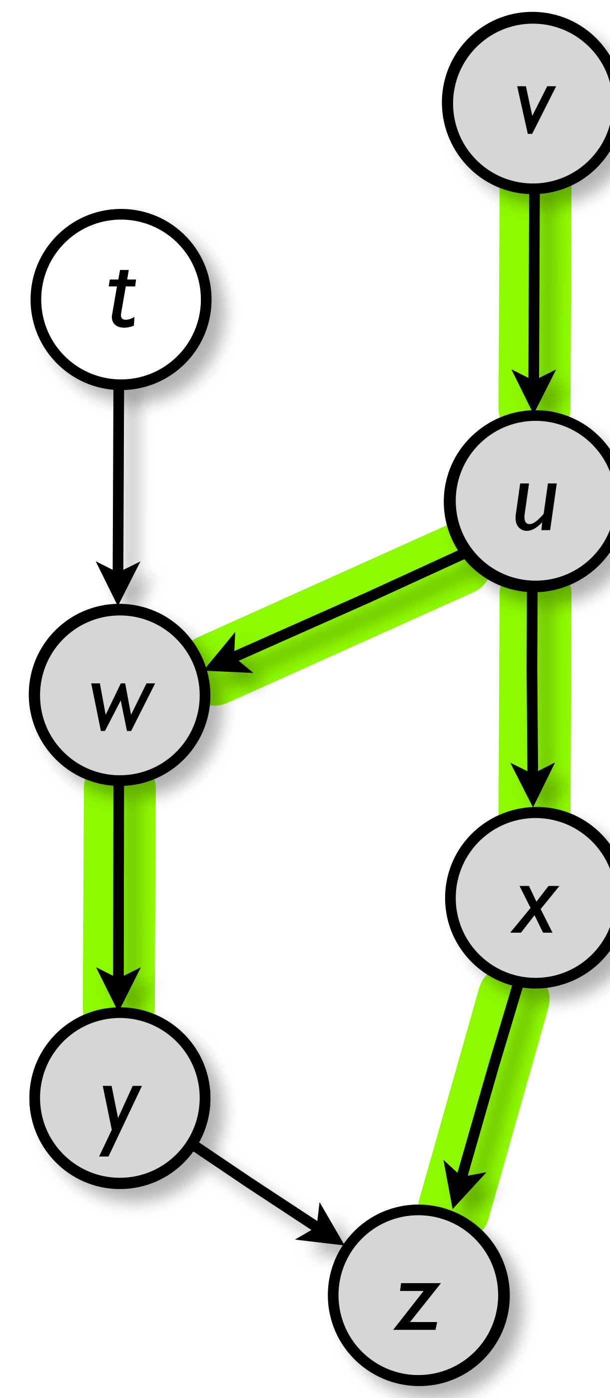
```
};
int const n_tasks = sizeof(tasks) / sizeof(char*);

struct bfs_counter
  : boost::default_bfs_visitor // inherit empty event point actions
{
    bfs_counter() : ecounter(0), vcounter(0) {}

    template <class Edge, class Graph>
    void tree_edge(Edge e, const Graph&)
    {
       std::cout << "Found tree edge #" << ecounter++ << "\n";
    }

    template <class Vertex, class Graph>
    void discover_vertex(Vertex u, const Graph&)
    {
       std::cout << "Found vertex #" << vcounter++ << "\n";
    }
private:
    std::size_t ecounter, vcounter;
};

int main()
{
```

```
};
int const n_tasks = sizeof(tasks) / sizeof(char*);

struct bfs_counter
  : boost::default_bfs_visitor // inherit empty event point actions
{
    bfs_counter() : ecounter(0), vcounter(0) {}

    template <class Edge, class Graph>
    void tree_edge(Edge e, const Graph&)
    {
       std::cout << "Found tree edge #" << ecounter++ << "\n";
    }

    template <class Vertex, class Graph>
    void discover_vertex(Vertex u, const Graph&)
    {
       std::cout << "Found vertex #" << vcounter++ << "\n";
    }
private:
    std::size_t ecounter, vcounter;
};

int main()
{
```

```
int main()
{
    using namespace boost;
    adjacency_list<listS, vecS, directedS> g(n_tasks);
    add_edge(0, 3, g);
    add_edge(1, 3, g);
    add_edge(1, 4, g);
    add_edge(2, 1, g);
    add_edge(3, 5, g);
    add_edge(4, 6, g);
    add_edge(5, 6, g);

    std::deque<int> topo_order;

    boost::topological_sort(
      g, std::front_inserter(topo_order),
      vertex_index_map(boost::identity_property_map()));

    for (std::deque<int>::iterator i = topo_order.begin();
         i != topo_order.end(); ++i)
    {
      std::cout << tasks[*i] << std::endl;
    }
}
```

```cpp
int main()
{
    using namespace boost;
    adjacency_list<listS, vecS, directedS> g(n_tasks);
    add_edge(0, 3, g);
    add_edge(1, 3, g);
    add_edge(1, 4, g);
    add_edge(2, 1, g);
    add_edge(3, 5, g);
    add_edge(4, 6, g);
    add_edge(5, 6, g);

    boost::breadth_first_search( g, 2, visitor( bfs_counter() ) );
}
```

```
$ ./x
Found vertex #0
Found tree edge #0
Found vertex #1
Found tree edge #1
Found vertex #2
Found tree edge #2
Found vertex #3
Found tree edge #3
Found vertex #4
Found tree edge #4
Found vertex #5
```



Depth-First Search

```
DFS(G)
    for each vertex u ∈ V[G]
        color[u] ← WHITE                  ◁ vis.initialize_vertex(u,G)
    for each vertex u ∈ V[G]
        if (color[v] = WHITE)
            call DFS-VISIT(G, u)

DFS-VISIT(G, u)
    color[u] ← GRAY                       ◁ vis.discover_vertex(u,G)
    for each v ∈ Adj[u]                   ◁ vis.examine_edge((u,v),G)
        if (color[v] = WHITE)             ◁ vis.tree_edge((u,v),G)
        ...
            call DFS-VISIT(G, u)
        else if (color[v] = GRAY)         ◁ vis.back_edge((u,v),G)
            ...
        else                              ◁ vis.cross_or_forward_edge((u,v),G)
            ...
    color[u] ← BLACK                      ◁ vis.finish_vertex(u,G)
```

All descendants
finished prior to this

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/breadth_first_search.hpp>

const char* tasks[] = {
    "pick up kids from school",              // 0
    "buy groceries (and snacks)",            // 1
    "get cash at ATM",                       // 2
    "drop off kids at soccer practice",      // 3
    "cook dinner",                           // 4
    "pick up kids from soccer",              // 5
    "eat dinner"                             // 6
};
int const n_tasks = sizeof(tasks) / sizeof(char*);

struct bfs_counter
  : boost::default_bfs_visitor // inherit empty event point actions
{
    bfs_counter() : ecounter(0), vcounter(0) {}

    template <class Edge, class Graph>
    void tree_edge(Edge e, const Graph&)
    {
       std::cout << "Found tree edge #" << ecounter++ << "\n";
    }
```

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/depth_first_search.hpp>

const char* tasks[] = {
    "pick up kids from school",             // 0
    "buy groceries (and snacks)",           // 1
    "get cash at ATM",                      // 2
    "drop off kids at soccer practice",     // 3
    "cook dinner",                          // 4
    "pick up kids from soccer",             // 5
    "eat dinner"                            // 6
};
int const n_tasks = sizeof(tasks) / sizeof(char*);

struct bfs_counter
  : boost::default_bfs_visitor // inherit empty event point actions
{
    bfs_counter() : ecounter(0), vcounter(0) {}

    template <class Edge, class Graph>
    void tree_edge(Edge e, const Graph&)
    {
       std::cout << "Found tree edge #" << ecounter++ << "\n";
    }
```

```
};
int const n_tasks = sizeof(tasks) / sizeof(char*);

struct bfs_counter
  : boost::default_bfs_visitor // inherit empty event point actions
{
    bfs_counter() : ecounter(0), vcounter(0) {}

    template <class Edge, class Graph>
    void tree_edge(Edge e, const Graph&)
    {
       std::cout << "Found tree edge #" << ecounter++ << "\n";
    }

    template <class Vertex, class Graph>
    void discover_vertex(Vertex u, const Graph&)
    {
       std::cout << "Found vertex #" << vcounter++ << "\n";
    }
private:
    std::size_t ecounter, vcounter;
};

int main()
{
```

```
};
int const n_tasks = sizeof(tasks) / sizeof(char*);

template <class OutputIterator>
struct topo_sort_visitor : boost::default_dfs_visitor
{
    topo_sort_visitor(OutputIterator iter)
      : m_iter(iter) { }

    template <class Vertex, class Graph>
    void finish_vertex(Vertex u, const Graph&)
    { *m_iter++ = u; }
private:
    OutputIterator m_iter;
};








int main()
{
```

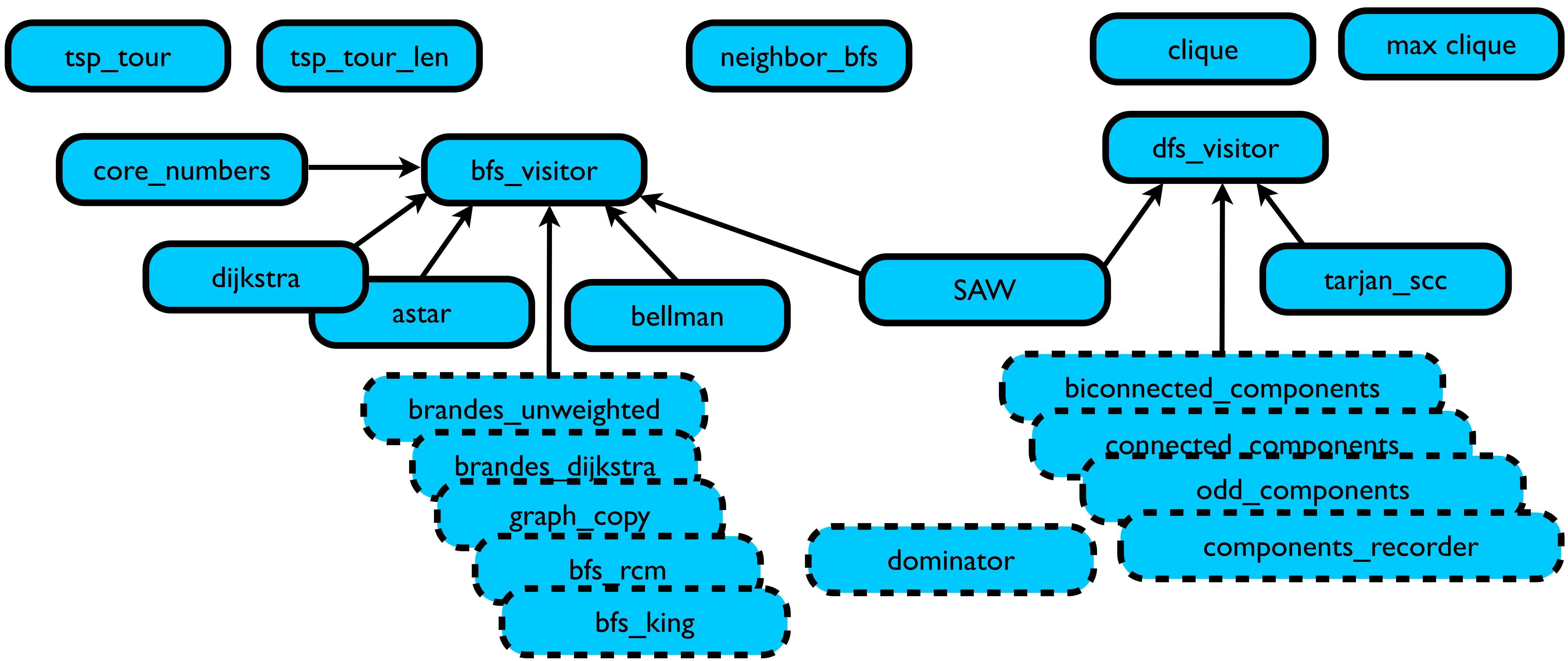
```
};
int const n_tasks = sizeof(tasks) / sizeof(char*);

template <class OutputIterator>
struct topo_sort_visitor : boost::default_dfs_visitor
{
    topo_sort_visitor(OutputIterator iter)
      : m_iter(iter) { }

    template <class Vertex, class Graph>
    void finish_vertex(Vertex u, const Graph&)
    { *m_iter++ = u; }
private:
    OutputIterator m_iter;
};






int main()
{
```

```
};
int const n_tasks = sizeof(tasks) / sizeof(char*);

template <class OutputIterator>
struct topo_sort_visitor : boost::default_dfs_visitor
{
    topo_sort_visitor(OutputIterator iter)
      : m_iter(iter) { }

    template <class Vertex, class Graph>
    void finish_vertex(Vertex u, const Graph&)
    { *m_iter++ = u; }
private:
    OutputIterator m_iter;
};

template <class Graph, class OutputIterator>
void topological_sort(Graph& g, OutputIterator result_iter)
{
    topo_sort_visitor<OutputIterator> vis(result_iter);
    boost::depth_first_search(g, boost::visitor(vis));
}

int main()
{
```

```cpp
int main()
{
    using namespace boost;
    adjacency_list<listS, vecS, directedS> g(n_tasks);
    add_edge(0, 3, g);
    add_edge(1, 3, g);
    add_edge(1, 4, g);
    add_edge(2, 1, g);
    add_edge(3, 5, g);
    add_edge(4, 6, g);
    add_edge(5, 6, g);

    boost::breadth_first_search( g, 2, visitor( bfs_counter() ) );
}
```

```
int main()
{
    using namespace boost;
    adjacency_list<listS, vecS, directedS> g(n_tasks);
    add_edge(0, 3, g);
    add_edge(1, 3, g);
    add_edge(1, 4, g);
    add_edge(2, 1, g);
    add_edge(3, 5, g);
    add_edge(4, 6, g);
    add_edge(5, 6, g);

    std::deque<int> topo_order;

    boost::topological_sort(
       g, std::front_inserter(topo_order),
       vertex_index_map(boost::identity_property_map()));

    for (std::deque<int>::iterator i = topo_order.begin();
         i != topo_order.end(); ++i)
    {
       std::cout << tasks[*i] << std::endl;
    }
}
```

BGL Visitor Models



boostpro computing

Named Parameters

Taming the Savage Beast

Dijkstra's Algorithm

Parameter	Role	Default
graph	in	
start vertex	in	
weight map	in	*interior*
vertex index map	in	*interior*
predecessor map	out	*discarded*
distance map	out	*discarded*
distance combine function	in	`closed_plus<D>()`
distance compare function	in	`std::less<D>()`
distance infinity constant	in	`std::numeric_limits<D>::max()`
distance zero constant	in	`D()`
color map	state	`vector<seminibble>(num_vertices(g))`
visitor	in	*NOP*

- Most parameters have useful defaults
- N explicit arguments required if Nth default unsuitable
- No parameter order can prevent wasted defaults

boostpro computing

Dijkstra's Algorithm/ Positional Parameters

```
typedef boost::graph_traits<G>::vertex_descriptor vertex;
std::map<vertex, vertex> p;
std::map<vertex, float> d;

boost::dijkstra_shortest_paths(
  g, start_vertex,
  boost::make_assoc_property_map(p),
  boost::make_assoc_property_map(d),
  get(boost::edge_weight, g), my_index_map,
  std::less<int>(), boost::closed_plus<int>(),
  std::numeric_limits<int>::max(),
  0, my_dijkstra_visitor()
);
```

boostpro computing

Dijkstra's Algorithm/ Named Parameters

```
boost::dijkstra_shortest_paths(
  g, start_vertex,
  boost::index_map(my_index_map)
    .visitor(my_dijkstra_visitor())
);
```

Note dot used for chaining

boostpro computing

Dijkstra's Algorithm/ Boost.Parameter

Coming Soon(?) to a Boost Release Near You

```
boost::dijkstra_shortest_paths(
  g, start_vertex,
  index_map = my_index_map,
  visitor = my_dijkstra_visitor()
);
```

Graph Generators: adjacency_list structure

Structure of each vertex's out-edge list

Structure of the "spine"

directedS or undirectedS

```
adjacency_list<
  OutEdgeList,
  VertexList,
  Directed,
  VertexProperty,
  EdgeProperty,
  GraphProperty,
  EdgeList
>
```

```
adjacency_matrix<
  Directed,
  VertexProperty,
  ...y,
  ...y,
```

Selector	Container
vecS	std::vector
listS	std::list
slistS	std/tr1::slist
setS	std::set
multisetS	std::multiset
hash_setS	std/tr1::unordered_set
hash_multisetS	std/tr1::unordered_multiset

Graph Generators: “bundled” properties

```
adjacency_list<
  OutEdgeList,
  VertexList,
  Directed,
  VertexProperty,
  EdgeProperty,
  GraphProperty,
  EdgeList
>
```

Arbitrary per-vertex type

Arbitrary per-edge type

Arbitrary per-graph type

```
adjacency_matrix<
  Directed,
  VertexProperty,
  EdgeProperty,
  GraphProperty,
  Allocator>
```

Interior Property Map Access: `get(map_id,g)`

- Maps from modern bundles
  - `map_id` is a bundle member pointer, e.g. `&EBundle::length`
  - or `vertex_bundle`/`edge_bundle` to access whole bundle
  - Pass explicitly, usually via named parameters:

    ```
    weight_map( get(&EdgeBundle::length, g) )
    ```
- Otherwise `map_id` is a predefined key such as `edge_weight`
- Algorithms use some predefined keys to build default property maps:
  - *implicit interior maps*, e.g. `vertex_index_map` of `adjacency_list<…,vecS>`
  - old-fashioned *"internal properties via property lists"*
- No special treatment of other old-fashioned internal properties (e.g. `vertex_name`)
- Yes, the overloading of `get(…)` for this purpose is confusing. Mea culpa.

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>

template <class G>
void populate(G& g)
{
    typename boost::graph_traits<G>::vertex_descriptor v = add_vertex(g);
    add_edge(v,v,g);
}

struct City
{
    std::string name;
    int population;
    std::vector<int> zipcodes;
};

struct Highway
{
    std::string name;
    double miles;
    int speed_limit;
    int lanes;
    bool divided;
};
```

```
struct City
{
    std::string name;
    int population;
    std::vector<int> zipcodes;
};

struct Highway
{
    std::string name;
    double miles;
    int speed_limit;
    int lanes;
    bool divided;
};

std::ostream& operator<<(std::ostream& o, Highway const& h)
{ return o << h.name << ": " << h.miles << " miles x "
          << h.lanes << " lanes @ " << h.speed_limit << " kph."; }

typedef boost::adjacency_list<
    boost::listS, boost::vecS,
    boost::bidirectionalS,
    City, Highway
> Graph;
```

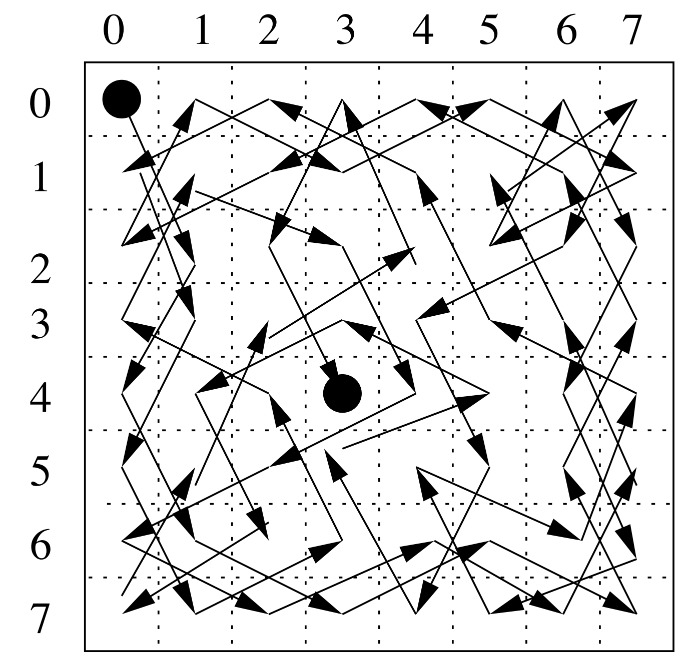
```
typedef boost::adjacency_list<
    boost::listS, boost::vecS,
    boost::bidirectionalS,
    City, Highway
> Graph;

int main()
{
    Graph g;
    populate(g); // create vertices and edges

    // Set properties on the first vertex
    Graph::vertex_descriptor v
    = *vertices(g).first;
    g[v].name = "Troy";
    g[v].population = 49170;
    g[v].zipcodes.push_back(12180);

    // Set properties on the first edge
    Graph::edge_descriptor e
    = *out_edges(v, g).first;
    g[e].name = "I-87";
    g[e].miles = 10;
    g[e].speed_limit = 65;
    g[e].lanes = 4;
```

```
    g[v].population = 49170;
    g[v].zipcodes.push_back(12180);

    // Set properties on the first edge
    Graph::edge_descriptor e
    = *out_edges(v, g).first;
    g[e].name = "I-87";
    g[e].miles = 10;
    g[e].speed_limit = 65;
    g[e].lanes = 4;
    g[e].divided = true;

    std::cout << get(boost::edge_bundle, g)[e] << std::endl;

    std::vector<double> distances(num_vertices(g));

    boost::dijkstra_shortest_paths(
      g, *vertices(g).first,
      weight_map( get(&Highway::miles, g) )
      .distance_map(
         make_iterator_property_map(
             distances.begin(), get(boost::vertex_index, g) )
      )
    );
}
```

Exercise

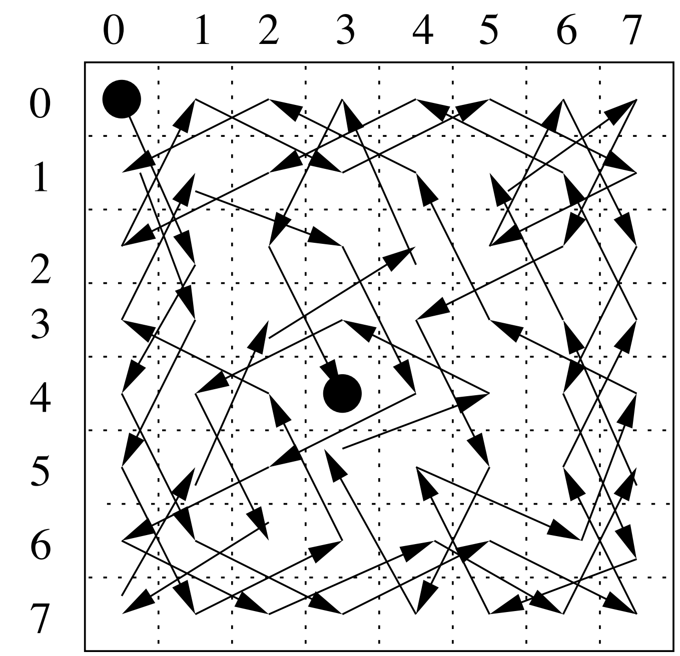
Implicit Graphs: Knight's Tour

Implicit Graphs



- A graph need not be a concrete data structure
- Inputs to algorithms need only model the necessary graph concepts
- Example: finding a path for a knight through a chessboard
  - "Knight's Tour Problem," finding a Hamiltonian path — is NP complete
  - Start with something easier

Your mission



- NxN chess board for arbitrary N
- Use BGL to find the shortest Knight's path from 0,0 to every other reachable position on the board
- Draw the resulting search tree for a 5x5 board
- Extra credit:
  - Implement a backtracking search for a full tour
  - Look up Warnsdorff's 1823 heuristic and apply it to your backtracking search.
  - How would your graph representation change for solving the real Knight's tour using, say, plain breadth-first search?