Boost.Graphの設計と
最短経路アルゴリズムの使い方いろいろ

株式会社ロングゲート
高橋 晶(Akira Takahashi)
faithandbrave@longgate.co.jp
2014/03/01(土) Boost.勉強会 #14 東京

はじめに

- この発表では、Boost Graph Libraryの設計をメインに話します。
- また、その設計をベースに作られた最短経路アルゴリズムを、どのようにして使うのかを話します。

話すこと

- Boost.Graphとグラフ理論の概要
- Boost.Graphの設計
  - データ構造とアルゴリズムの分離
  - グラフコンセプト
  - プロパティマップ
- 最短経路アルゴリズムの使い方
  - 名前付き引数
  - イベントビジター
  - 動的プロパティマップ

Chapter 1

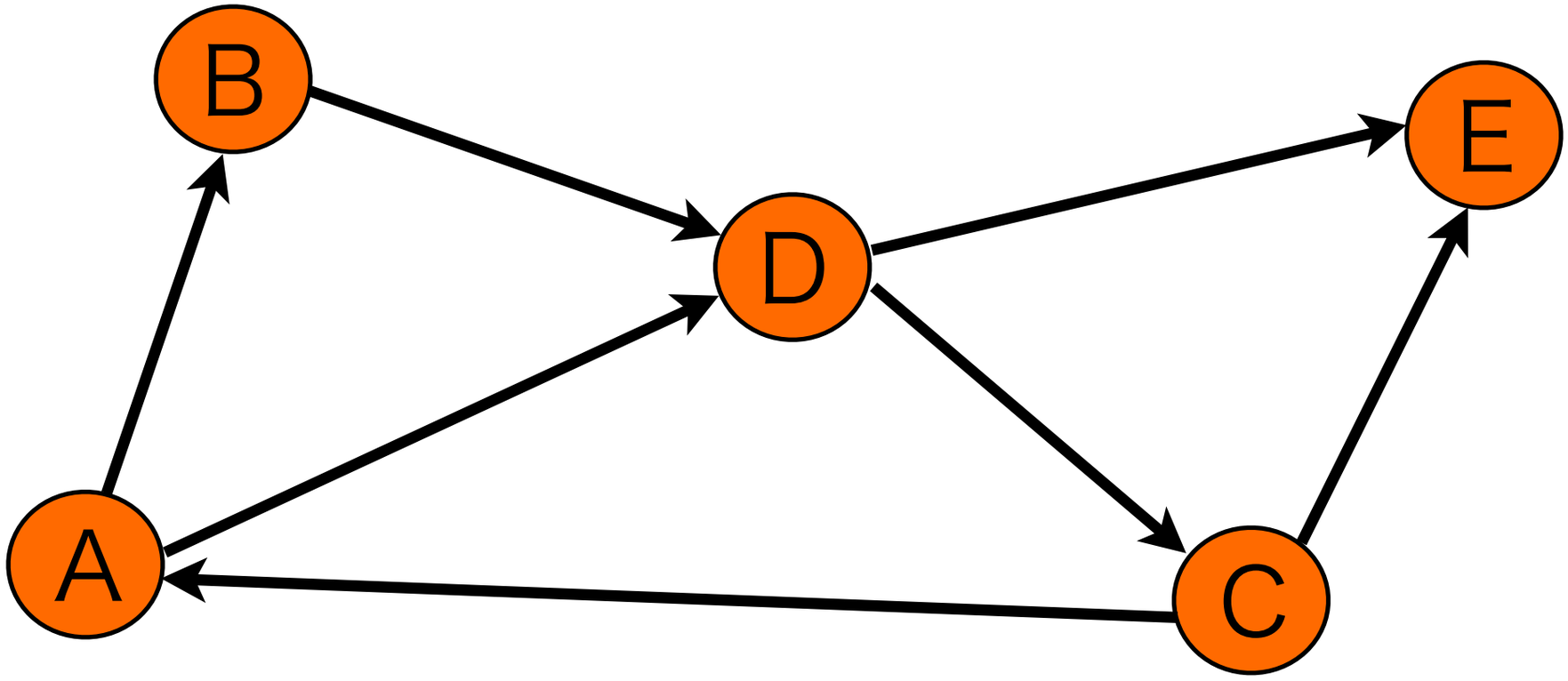
Boost.Graphとグラフ理論の概要

Boost.Graphとは

- グラフのデータ構造とアルゴリズムのライブラリ。
- STLの概念に基いて、データ構造とアルゴリズムの分離を行っているのが特徴。
- この設計は、後発の多くのグラフライブラリに、大きな影響を与えた。

グラフとは

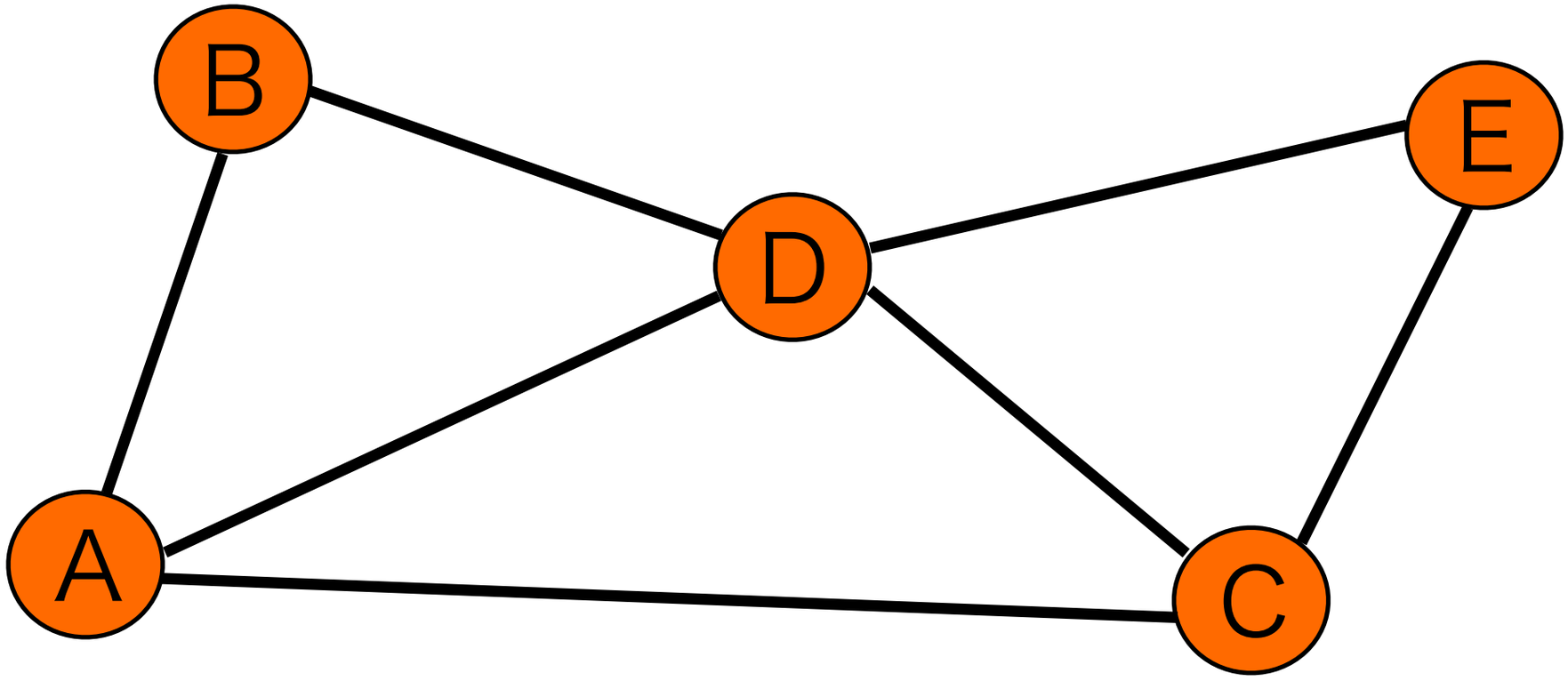
- 頂点(vertex)と辺(edge)からなるデータ構造
  - 棒グラフや折れ線グラフとは関係ない



辺が方向を持つかどうか

- グラフには、辺が方向という情報を持つかどうかで、**有向グラフ(directed)**、**無向グラフ(undirected)**の2つの分類がある。

- 前ページのは有向グラフ、以下は無向グラフ。



どんなものがグラフか

- 電車の路線
- 道路
- ネットワーク
- ファイルの依存関係
- クラス図
- 行列
  - チェス盤
- Twitterのフォロー関係や、情報の広がり方

グラフを使ってどんなことをするか

- 最短経路を求める
- 一筆書きを求める
- ネットワーク分析
    - コミュニティや、情報を広げる中心にいるのは誰か
    - コミュニティ同士をつなぐ架け橋、またはボトルネックは誰か
- ページランクの算出(Google検索)
- 迷路を解く

グラフ理論関係のオススメ本

『**最短経路の本**』
グラフ理論の入門書。
最短経路、スパニングツリー、
オイラー路等。

『**オープンソースで学ぶ
社会ネットワーク分析**』
グラフ理論の入門から
ネットワーク分析まで。

Chapter 2

Boost.Graphの設計

グラフ型

- Boost.Graphでは、以下のグラフ型を提供している：
  - `adjacency_list`：隣接リスト
  - `adjacency_matrix`：隣接行列
  - `compressed_sparse_row_graph`：CSRグラフ

グラフ型への操作

- Boost.Graphでは、グラフ型への統一的な操作を提供している。
- グラフ型が直接持つインタフェース(メンバ)を使うのではなく、オーバーロード可能なフリー関数を介してグラフを操作する。

```
template <class Graph>
void proc_graph(Graph& g)
{
  // 頂点のシーケンスを取得して走査
  for (const auto& vertex : vertices(g)) { … }

  // 辺のシーケンスを取得して走査
  for (const auto& edge : edges(g)) { … }
}
```

vertices()/edges()はイテレータの組を返すため、範囲for文で使用するには、変換・アダプトが必要

グラフ型への操作

- これらのフリー関数を使用したアルゴリズムには、Boost.Graphのコンセプトを満たす、あらゆるグラフ型を適用できる。
- Boost.Graphのグラフ型はもちろん扱えて、

```
adjacency_list<> g;
proc_graph(g);
```

グラフ型への操作

- これらのフリー関数を使用したアルゴリズムには、Boost.Graphのコンセプトを満たす、あらゆるグラフ型を適用できる。
- Boost.Graphのグラフ型はもちろん扱えて、

```
compressed_sparse_row_graph<> g;
proc_graph(g);
```

グラフ型への操作

- これらのフリー関数を使用したアルゴリズムには、Boost.Graphのコンセプトを満たす、あらゆるグラフ型を適用できる。

- `std::vector`もグラフ型として使用できる。

```
std::vector<std::list<int>> g;
proc_graph(g);
```

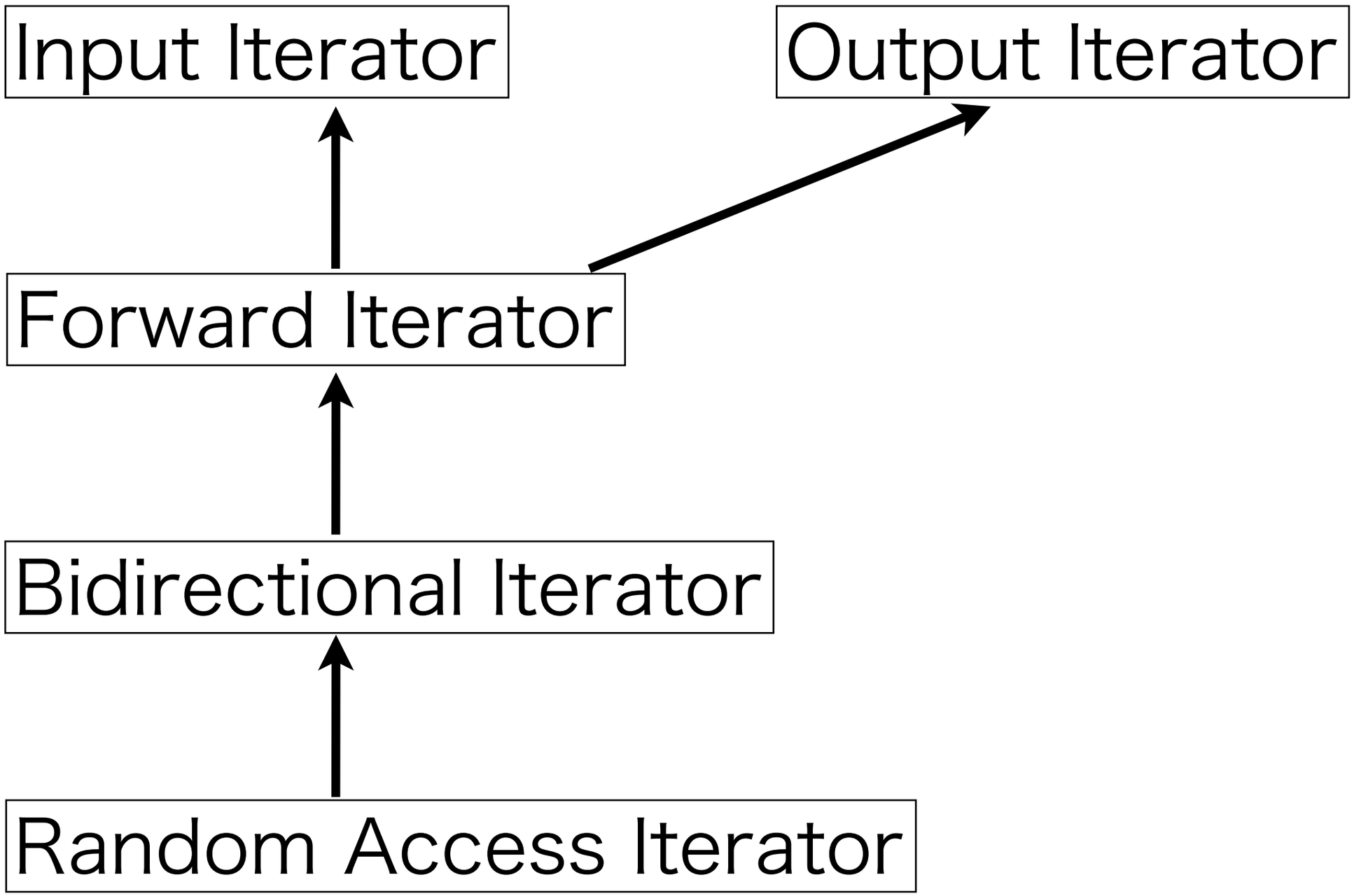
Boost.GraphとSTL

- Alexander Stepanovが設計したSTLは、コンテナとアルゴリズムを、**イテレータという中間インタフェース**を介することで分離する、というものだった。

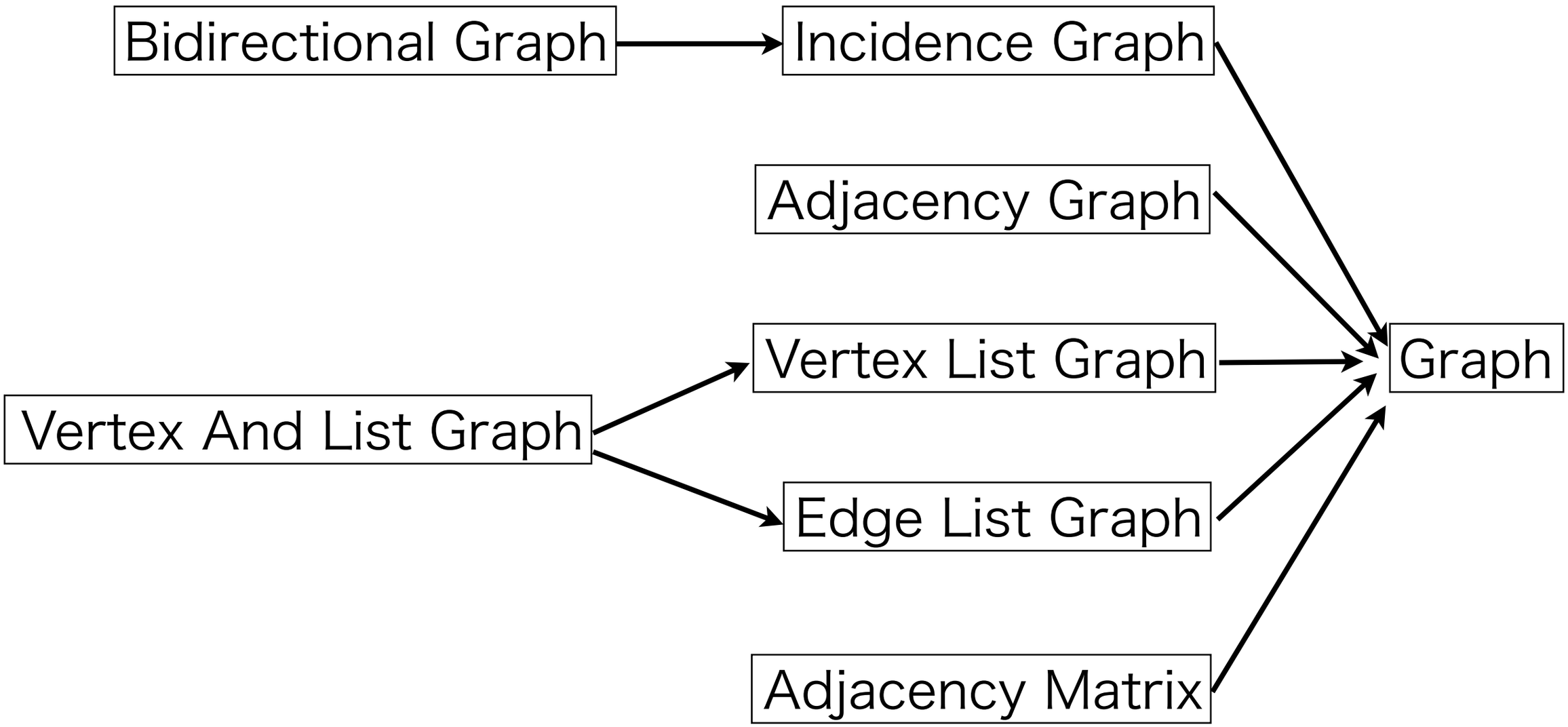
- Boost.Graphは、`vertices()`や`edges()`といった**グラフコンセプトのフリー関数**を、グラフ構造とアルゴリズムの中間インタフェースとしている。

- Boost.Graphが定義するグラフコンセプトのフリー関数を持つ型であれば、なんでもBoost.Graphのアルゴリズムを使用できる。

STLのイテレータコンセプト



Boost.Graphのグラフコンセプト



具体的にどのコンセプトで何の機能が使えるかは、
ドキュメントを参照！

プロパティマップ

- グラフには、様々な種類のデータが付加される。
  路線を表すなら、以下のような情報が必要だろう：
  - 頂点には「駅名」
  - 辺には「駅間の距離」
  - グラフ全体には「路線名」
- Boost.Graphでは、グラフの各要素に任意のデータを持たせる**プロパティマップ**という仕組みが用意されている。

adjacency_listのテンプレートパラメータ

- デフォルトで使用するグラフ型adjacency_listには、以下のテンプレートパラメータがある。

```
template <class OutEdgeListS = vecS,
          class VertexListS = vecS,
          class DirectedS = directedS,
          class VertexProperties = no_property,
          class EdgeProperties = no_property,
          class GraphProperties = no_property,
          class EdgeListS = listS>
class adjacency_list;
```

adjacency_listのテンプレートパラメータ

```
template <class OutEdgeListS = vecS,
          class VertexListS = vecS,
          class DirectedS = directedS,
          class VertexProperties = no_property,
          class EdgeProperties = no_property,
          class GraphProperties = no_property,
          class EdgeListS = listS>
class adjacency_list;
```

- グラフの隣接構造(入辺と出辺)を表すためのコンテナ
  - vecS：std::vector
  - listS：std::list
  - setS：std::set

adjacency_listのテンプレートパラメータ

```
template <class OutEdgeListS = vecS,
          class VertexListS = vecS,
          class DirectedS = directedS,
          class VertexProperties = no_property,
          class EdgeProperties = no_property,
          class GraphProperties = no_property,
          class EdgeListS = listS>
class adjacency_list;
```

- グラフの頂点集合を表すためのコンテナ
  - vecS：std::vector
  - listS：std::list
  - setS：std::set

adjacency_listのテンプレートパラメータ

```
template <class OutEdgeListS = vecS,
          class VertexListS = vecS,
          class DirectedS = directedS,
          class VertexProperties = no_property,
          class EdgeProperties = no_property,
          class GraphProperties = no_property,
          class EdgeListS = listS>
class adjacency_list;
```

- 有向グラフか無向グラフかを選択する。
  - directedS : 有向
  - undirectedS : 無向
  - bidirectionalS : 双方向(有向の辺が2本)

adjacency_listのテンプレートパラメータ

```
template <class OutEdgeListS = vecS,
          class VertexListS = vecS,
          class DirectedS = directedS,
          class VertexProperties = no_property,
          class EdgeProperties = no_property,
          class GraphProperties = no_property,
          class EdgeListS = listS>
class adjacency_list;
```

- それぞれ、頂点、辺、グラフのプロパティを指定する。

adjacency_listのテンプレートパラメータ

```
adjacency_list<
  vecS,
  vecS,
  directedS,
  no_property,
  property<edge_weight_t, int>
>;
```

- 辺に「距離」を持たせる

adjacency_listのテンプレートパラメータ

```
adjacency_list<
  vecS,
  vecS,
  directedS,
  no_property,
  property<edge_weight_t, int,
    property<edge_name_t, std::string>>
>;
```

- 辺に「距離」と「名前」を持たせる

プロパティの取得と設定

```
int weight = boost::get(edge_weight, // これはタグ
                        graph,
                        edge_desc);
```

- プロパティの取得にはboost::get()を使う。

プロパティの取得と設定

```
boost::put(edge_weight, // これはタグ
           graph,
           edge_desc,
           weight_value);
```

- プロパティの取得にはboost::put()を使う。

ここまでのまとめ

- Boost.Graphは、STLの概念に基いて、データ構造とアルゴリズムの分離を行っている。
- STLのイテレータコンセプトと同じように、Boost.Graphもグラフコンセプトを持っている。
- グラフ構造には、ユーザーが様々な情報を付加する場合もある。そのために、プロパティマップという仕組みがある。

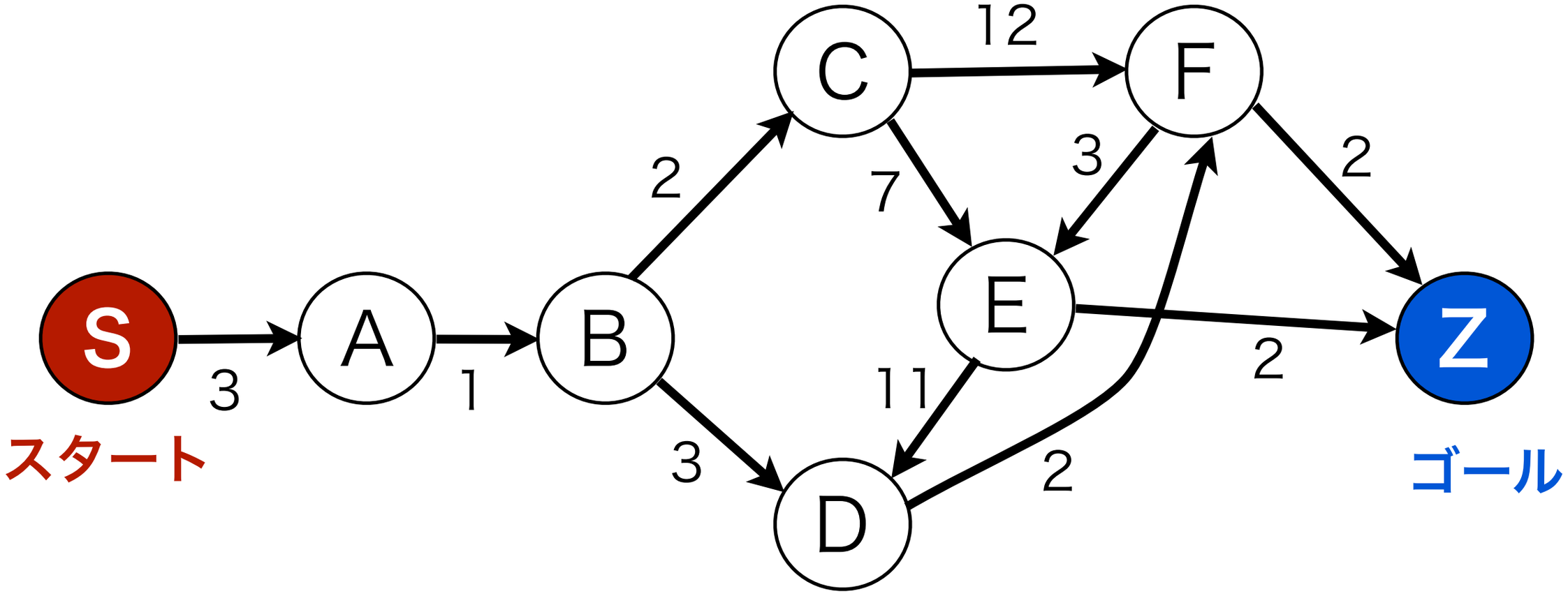
Chapter 3

最短経路アルゴリズムの使い方いろいろ

最短経路アルゴリズム

- ここでは、Boost.Graphに複数ある最短経路アルゴリズムの中から、dijkstra_shortest_paths()に絞って、いろいろな使い方を見ていきます。
- この関数は、ダイクストラ法による最短経路アルゴリズムです。ある頂点から全ての頂点への最短経路を求めます。

最短経路を求める有向グラフ



数字は距離を表す。(見た目と値が一致してないけど)

基本的な使い方

```
// 頂点を定義
enum { S, A, B, C, D, E, F, Z, N };
const std::string Names = "SABCDEFZ";

// グラフ型
using Graph = adjacency_list<
    listS,
    vecS,
    directedS,
    no_property,
    property<edge_weight_t, int>
>;
using Vertex = graph_traits<Graph>::vertex_descriptor;
```

基本的な使い方

```
const Graph g = make_my_graph(); // グラフデータを読み込む
const Vertex from = vertex(S, g); // 開始地点
const Vertex to   = vertex(Z, g); // 目的地

// 最短経路を計算
std::vector<Vertex> parents(boost::num_vertices(g));
boost::dijkstra_shortest_paths(g, from,
                boost::predecessor_map(&parents[0]));
```

基本的な使い方

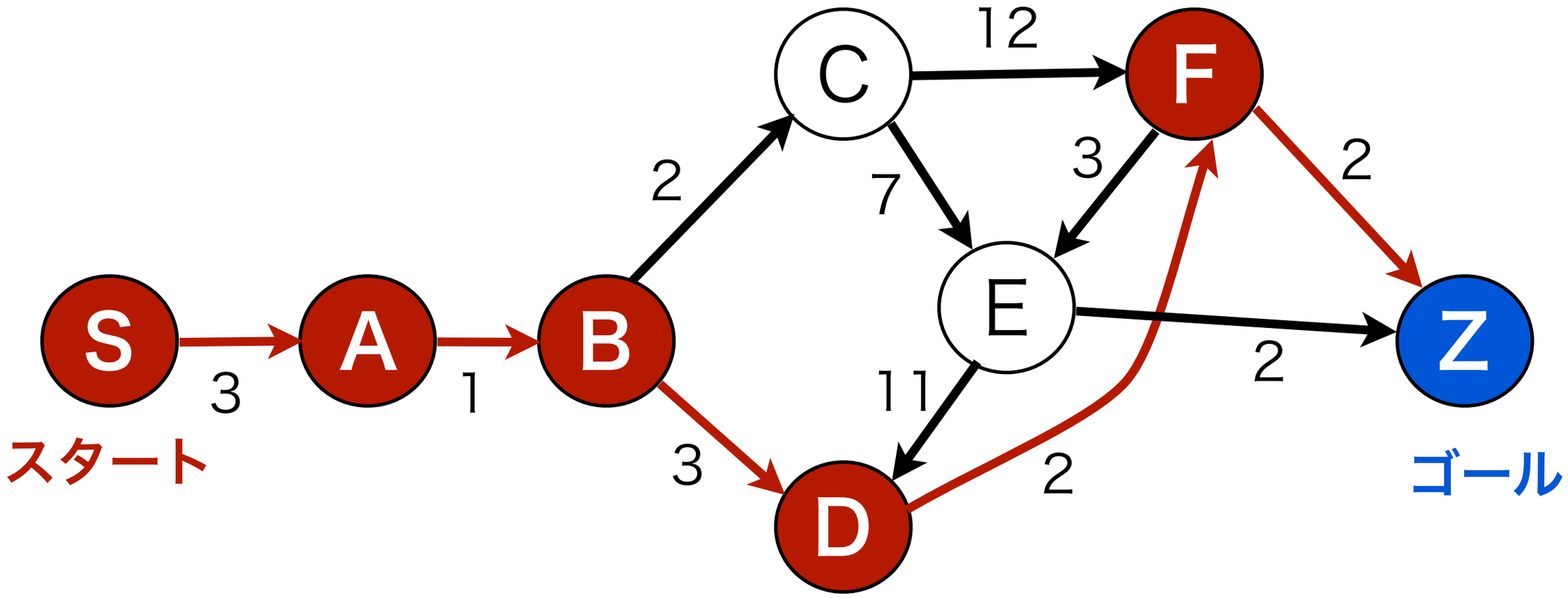
```
// 経路なし
if (parents[to] == to)
    return;

// 最短経路の頂点リストを作成
std::deque<Vertex> route;
for (Vertex v = to; v != from; v = parents[v]) {
    route.push_front(v);
}
route.push_front(from);

// 最短経路を出力
for (const Vertex v : route) {
    cout << Names[get(boost::vertex_index, g, v)] << endl;
}
```

基本的な使い方

- 最短経路が出力されました。

```
S
A
B
D
F
Z
```

最短経路



先行マップ

- `dijkstra_shortest_paths()`には、3つの引数を渡す。

1. グラフオブジェクト
2. 始点となる頂点
3. **先行マップ**

```
std::vector<Vertex> parents(boost::num_vertices(g));
boost::dijkstra_shortest_paths(g, from,
                boost::predecessor_map(&parents[0]));
```

先行マップ

- 先行マップは、求められた最短経路の木において、「**ある頂点の前に通る頂点**」を集めた辞書。
- 先行マップに目的地Zを与えると、Zの前を通る頂点Fが取得できる。
- これを開始頂点まで繰り返すことで、**逆順の最短経路**を頂点のリストとして取得できる。

```
// 最短経路の頂点リストを作成
std::deque<Vertex> route;
for (Vertex v = to; v != from; v = parents[v]) {
    route.push_front(v);
}
route.push_front(from);
```

名前付き引数

- 先行マップは、dijkstra_shortest_paths()にboost::predecessor_map()という関数を通して渡していた。
- これは名前付き引数。
  - 指定順に依存せず、引数名を指定して値を渡す方法。

```
std::vector<Vertex> parents(boost::num_vertices(g));
boost::dijkstra_shortest_paths(g, from,
                boost::predecessor_map(&parents[0]));
```

名前付き引数

- 名前付き引数ではないバージョンの場合は、
  その他多くのパラメータを指定する必要がある。

- 大変なので、名前付き引数バージョンを使おう。

```
boost::dijkstra_shortest_paths(g, from,
    predecessor_map,
    distance_map,
    weight_map,
    index_map,
    compare,
    combine,
    dist_inf,
    dist_zero,
    visitor
);
```

動的な距離を使用する

- ここまで、グラフオブジェクトに距離データを持たせていた。
- 距離を別で指定したい場合は、`dijkstra_shortest_paths()`に`weight_map`を渡す。

```
// 全ての頂点間の距離が1
boost::static_property_map<int> weight(1);

boost::dijkstra_shortest_paths(g, from,
                boost::predecessor_map(&parents[0]).
                weight_map(weight));
```

動的な距離を使用する

- 関数が呼ばれるたびに距離を計算した場合は、function_property_mapを使用する。

```
int f(EdgeDesc e) // 辺の距離を求める
{
  // source(e, g)とtarget(e, g)で2頂点を取得できるよ
  …
}

boost::dijkstra_shortest_paths(g, from,
   boost::predecessor_map(&parents[0]).
   weight_map(
     boost::make_function_property_map<EdgeDesc>(f)));
```

最短経路の距離を求める

- 最短経路全体でどれくらいの距離があるか知りたい場合は、distance_mapを指定する。

```
std::vector<int> distance(boost::num_vertices(g));
boost::dijkstra_shortest_paths(g, from,
   boost::predecessor_map(&parents[0]).
   distance_map(&distance[0]));

// 開始地点から目的地までの、距離の合計を取得
int d = distance[to];
```

イベントビジター

- Boost.Graphのアルゴリズムには、「**イベントビジター**」という仕組みがある。
- アルゴリズムの各ポイントで、任意の関数を呼び出せるというもの。

```
struct MyVisitor : dijkstra_visitor<> {
  template <class Vertex, class Graph>
  void discover_vertex(Vertex v, const Graph&)
  { …頂点を通ったら呼ばれるイベント… }
};

boost::dijkstra_shortest_paths(g, from,
            boost::predecessor_map(&parents[0]).
            visitor(MyVisitor));
```

イベントビジターは何に使うか

- 特定の頂点が見つかったら例外を投げて、アルゴリズムのループを脱出する。(A*探索で使う)
- 既存のアルゴリズムを使って、新たなアルゴリズムを作る。
  - `dijkstra_shortest_paths()`は、幅優先探索をイベントビジターでラップして作られている。
- コルーチンでアルゴリズムを中断する。
- アルゴリズム計算状況の可視化。

最短経路アルゴリズムまとめ

- Boost.Graphの最短経路アルゴリズムは、**先行マップ**によって経路を取得する。
- Boost.Graphの最短経路アルゴリズムは、指定する**名前付き引数**の種類によって、いろいろな結果を取得できる。
- 辺の距離は、グラフに持たせることもできるし、アルゴリズムに別途渡すこともできる。
- Boost.Graphのアルゴリズムは**イベントビジター**という仕組みによって、任意のポイントに関数を埋め込める。

アルゴリズムの設計

- C++Now! 2012のVoronoiに関する発表概要で、このような文章がある。

> ユーザビリティとは、その分野に精通していないユーザーにとっての、公開されているアルゴリズムインターフェースのわかりやすさである。同時に、精通しているユーザーにとっての、アルゴリズムを構成できる幅のことでもある。

- Boost.Graphのアルゴリズムはまさに、この考え方に基いて設計されていると言える。

全体まとめ

- Boost.Graphは、STLの概念に基いて、グラフのデータ構造とアルゴリズムを分離している。
- Boost.Graphのアルゴリズムは、ひとつのアルゴリズムを複数の目的に利用できる。
- グラフを普段から使っている人にとっては、Boost.Graphのユーザーガイドとして。
  そうでない人にとっては、ライブラリ設計のガイドとしてこの発表が役立てば幸いです。