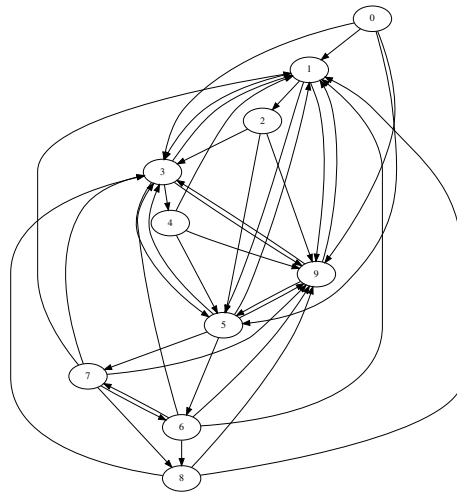


A well-connected C++11 Boost.Graph tutorial

Richèl Bilderbeek

February 27, 2016



Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 10 |
| 1.1 | Why this tutorial | 10 |
| 1.2 | Tutorial style | 11 |
| 1.3 | Code snippets | 11 |
| 1.4 | Coding style | 12 |
| 1.5 | License | 12 |
| 1.6 | Feedback | 13 |
| 1.7 | Acknowledgements | 13 |
| 1.8 | Outline | 13 |
| 2 | Building graphs without properties | 14 |
| 2.1 | Creating an empty (directed) graph | 18 |
| 2.2 | Creating an empty undirected graph | 19 |
| 2.3 | Counting the number of vertices | 21 |
| 2.4 | Counting the number of edges | 22 |
| 2.5 | Adding a vertex | 23 |

| | | |
|----------|---|-----------|
| 2.6 | Vertex descriptors | 24 |
| 2.7 | Get the vertex iterators | 25 |
| 2.8 | Get all vertex descriptors | 26 |
| 2.9 | Add an edge | 27 |
| 2.10 | boost::add_edge result | 29 |
| 2.11 | Getting the edge iterators | 29 |
| 2.12 | Edge descriptors | 30 |
| 2.13 | Get all edge descriptors | 31 |
| 2.14 | Creating a directed graph | 32 |
| 2.14.1 | Graph | 32 |
| 2.14.2 | Function to create such a graph | 33 |
| 2.14.3 | Creating such a graph | 33 |
| 2.14.4 | The .dot file produced | 34 |
| 2.14.5 | The .svg file produced | 34 |
| 2.15 | Creating K_2 , a fully connected undirected graph with two vertices | 35 |
| 2.15.1 | Graph | 35 |
| 2.15.2 | Function to create such a graph | 35 |
| 2.15.3 | Creating such a graph | 36 |
| 2.15.4 | The .dot file produced | 37 |
| 2.15.5 | The .svg file produced | 37 |
| 2.16 | ► Creating K_3 , a fully connected undirected graph with three vertices | 38 |
| 2.16.1 | Graph | 38 |
| 2.16.2 | Function to create such a graph | 38 |
| 2.16.3 | Creating such a graph | 39 |
| 2.16.4 | The .dot file produced | 39 |
| 2.16.5 | The .svg file produced | 40 |
| 2.17 | ► Creating a path graph | 40 |
| 2.17.1 | Graph | 40 |
| 2.17.2 | Function to create such a graph | 41 |
| 2.17.3 | Creating such a graph | 41 |
| 2.17.4 | The .dot file produced | 42 |
| 2.17.5 | The .svg file produced | 42 |
| 2.18 | ► Creating a Peterson graph | 43 |
| 2.18.1 | Graph | 43 |
| 2.18.2 | Function to create such a graph | 44 |
| 2.18.3 | Creating such a graph | 46 |
| 2.18.4 | The .dot file produced | 46 |
| 2.18.5 | The .svg file produced | 47 |
| 3 | Working on graphs without properties | 48 |
| 3.1 | Getting the vertices' out degree | 49 |
| 3.2 | ► Is there an edge between two vertices? | 50 |
| 3.3 | ► Get the edge between two vertices | 51 |
| 3.4 | ► Create a direct-neighbour subgraph from a vertex descriptor | 53 |

| | | |
|----------|---|-----------|
| 3.5 | ► Creating all direct-neighbour subgraphs from a graph without properties | 55 |
| 3.6 | ► Are two graphs isomorphic? | 57 |
| 3.7 | Saving a graph to a .dot file | 58 |
| 3.8 | Loading a directed graph from a .dot | 59 |
| 3.9 | Loading an undirected graph from a .dot file | 60 |
| 4 | Building graphs with named vertices | 62 |
| 4.1 | Creating an empty directed graph with named vertices | 63 |
| 4.2 | Creating an empty undirected graph with named vertices | 64 |
| 4.3 | Add a vertex with a name | 65 |
| 4.4 | Getting the vertices' names | 67 |
| 4.5 | Creating a Markov chain with named vertices | 69 |
| 4.5.1 | Graph | 69 |
| 4.5.2 | Function to create such a graph | 70 |
| 4.5.3 | Creating such a graph | 70 |
| 4.5.4 | The .dot file produced | 71 |
| 4.5.5 | The .svg file produced | 71 |
| 4.6 | Creating K_2 with named vertices | 72 |
| 4.6.1 | Graph | 72 |
| 4.6.2 | Function to create such a graph | 72 |
| 4.6.3 | Creating such a graph | 73 |
| 4.6.4 | The .dot file produced | 74 |
| 4.6.5 | The .svg file produced | 74 |
| 4.7 | ► Creating K_3 with named vertices | 75 |
| 4.7.1 | Graph | 75 |
| 4.7.2 | Function to create such a graph | 75 |
| 4.7.3 | Creating such a graph | 76 |
| 4.7.4 | The .dot file produced | 77 |
| 4.7.5 | The .svg file produced | 77 |
| 4.8 | ► Creating a path graph with named vertices | 78 |
| 4.8.1 | Graph | 78 |
| 4.8.2 | Function to create such a graph | 78 |
| 4.8.3 | Creating such a graph | 79 |
| 4.8.4 | The .dot file produced | 80 |
| 4.8.5 | The .svg file produced | 80 |
| 4.9 | ► Creating a Petersen graph with named vertices | 81 |
| 4.9.1 | Graph | 81 |
| 4.9.2 | Function to create such a graph | 82 |
| 4.9.3 | Creating such a graph | 84 |
| 4.9.4 | The .dot file produced | 84 |
| 4.9.5 | The .svg file produced | 85 |

| | | |
|----------|---|------------|
| 5 | Working on graphs with named vertices | 86 |
| 5.1 | Check if there exists a vertex with a certain name | 87 |
| 5.2 | Find a vertex by its name | 89 |
| 5.3 | Get a (named) vertex its degree, in degree and out degree | 90 |
| 5.4 | Get a vertex its name from its vertex descriptor | 92 |
| 5.5 | Set a (named) vertex its name from its vertex descriptor | 94 |
| 5.6 | Setting all vertices' names | 95 |
| 5.7 | Clear the edges of a named vertex | 96 |
| 5.8 | Remove a named vertex | 98 |
| 5.9 | ► Adding an edge between two named vertices | 99 |
| 5.10 | ► Removing the edge between two named vertices | 101 |
| 5.11 | ► Count the vertices with a certain name | 103 |
| 5.12 | ► Create a direct-neighbour subgraph from a vertex descriptor of a graph with named vertices | 105 |
| 5.13 | ► Creating all direct-neighbour subgraphs from a graph with named vertices | 107 |
| 5.14 | ► Are two graphs with named vertices isomorphic? | 109 |
| 5.15 | Saving an directed/undirected graph with named vertices to a .dot file | 112 |
| | 5.15.1 Using boost::make_label_writer | 112 |
| | 5.15.2 Using a C++11 lambda function | 113 |
| | 5.15.3 Demonstration | 115 |
| 5.16 | Loading a directed graph with named vertices from a .dot | 115 |
| 5.17 | Loading an undirected graph with named vertices from a .dot | 117 |
| 6 | Building graphs with named edges and vertices | 119 |
| 6.1 | Creating an empty directed graph with named edges and vertices | 120 |
| 6.2 | Creating an empty undirected graph with named edges and vertices | 122 |
| 6.3 | Adding a named edge | 123 |
| 6.4 | Adding a named edge between vertices | 125 |
| 6.5 | Getting the edges' names | 127 |
| 6.6 | Creating Markov chain with named edges and vertices | 129 |
| | 6.6.1 Graph | 129 |
| | 6.6.2 Function to create such a graph | 130 |
| | 6.6.3 Creating such a graph | 131 |
| | 6.6.4 The .dot file produced | 132 |
| | 6.6.5 The .svg file produced | 132 |
| 6.7 | Creating K_2 with named edges and vertices | 132 |
| | 6.7.1 Graph | 132 |
| | 6.7.2 Function to create such a graph | 133 |
| | 6.7.3 Creating such a graph | 134 |
| | 6.7.4 The .dot file produced | 135 |
| | 6.7.5 The .svg file produced | 136 |
| 6.8 | Creating K_3 with named edges and vertices | 136 |
| | 6.8.1 Graph | 136 |
| | 6.8.2 Function to create such a graph | 136 |

| | | |
|----------|---|------------|
| 6.8.3 | Creating such a graph | 137 |
| 6.8.4 | The .dot file produced | 139 |
| 6.8.5 | The .svg file produced | 139 |
| 6.9 | ► Creating a path graph with named edges and vertices | 139 |
| 6.9.1 | Graph | 140 |
| 6.9.2 | Function to create such a graph | 140 |
| 6.9.3 | Creating such a graph | 142 |
| 6.9.4 | The .dot file produced | 142 |
| 6.9.5 | The .svg file produced | 142 |
| 6.10 | ► Creating a Petersen graph with named edges and vertices | 143 |
| 6.10.1 | Graph | 143 |
| 6.10.2 | Function to create such a graph | 144 |
| 6.10.3 | Creating such a graph | 146 |
| 6.10.4 | The .dot file produced | 146 |
| 6.10.5 | The .svg file produced | 147 |
| 7 | Working on graphs with named edges and vertices | 148 |
| 7.1 | Check if there exists an edge with a certain name | 149 |
| 7.2 | Find an edge by its name | 150 |
| 7.3 | Get a (named) edge its name from its edge descriptor | 152 |
| 7.4 | Set a (named) edge its name from its edge descriptor | 153 |
| 7.5 | Removing the first edge with a certain name | 155 |
| 7.6 | ► Create a direct-neighbour subgraph from a vertex descriptor of a graph with named edges and vertices | 157 |
| 7.7 | ► Creating all direct-neighbour subgraphs from a graph with named edges and vertices | 159 |
| 7.8 | Saving an undirected graph with named edges and vertices as a .dot | 161 |
| 7.9 | Loading a directed graph with named edges and vertices from a .dot | 163 |
| 7.10 | Loading an undirected graph with named edges and vertices from a .dot | 165 |
| 8 | Building graphs with bundled vertices | 168 |
| 8.1 | Creating the bundled vertex class | 168 |
| 8.2 | Create the empty directed graph with bundled vertices | 170 |
| 8.3 | Create the empty undirected graph with bundled vertices | 171 |
| 8.4 | Add a bundled vertex | 171 |
| 8.5 | Getting the bundled vertices' my_vertexes | 172 |
| 8.6 | Creating a two-state Markov chain with bundled vertices | 173 |
| 8.6.1 | Graph | 173 |
| 8.6.2 | Function to create such a graph | 173 |
| 8.6.3 | Creating such a graph | 174 |
| 8.6.4 | The .dot file produced | 175 |
| 8.6.5 | The .svg file produced | 177 |
| 8.7 | Creating K_2 with bundled vertices | 178 |

| | | |
|-----------|---|------------|
| 8.7.1 | Graph | 178 |
| 8.7.2 | Function to create such a graph | 179 |
| 8.7.3 | Creating such a graph | 179 |
| 8.7.4 | The .dot file produced | 180 |
| 8.7.5 | The .svg file produced | 182 |
| 9 | Working on graphs with bundled vertices | 183 |
| 9.1 | Has a bundled vertex with a my_bundled_vertex | 183 |
| 9.2 | Find a bundled vertex with a certain my_bundled_vertex | 185 |
| 9.3 | Get a bundled vertex its 'my_bundled_vertex' | 187 |
| 9.4 | Set a bundled vertex its my_vertex | 188 |
| 9.5 | Setting all bundled vertices' my_vertex objects | 189 |
| 9.6 | Storing a graph with bundled vertices as a .dot | 190 |
| 9.7 | Loading a directed graph with bundled vertices from a .dot | 193 |
| 9.8 | Loading an undirected graph with bundled vertices from a .dot | 196 |
| 10 | Building graphs with bundled edges and vertices | 198 |
| 10.1 | Creating the bundled edge class | 199 |
| 10.2 | Create an empty directed graph with bundled edges and vertices | 201 |
| 10.3 | Create an empty undirected graph with bundled edges and vertices | 202 |
| 10.4 | Add a bundled edge | 203 |
| 10.5 | Getting the bundled edges my_edges | 205 |
| 10.6 | Creating a Markov-chain with bundled edges and vertices | 206 |
| 10.6.1 | Graph | 206 |
| 10.6.2 | Function to create such a graph | 207 |
| 10.6.3 | Creating such a graph | 209 |
| 10.6.4 | The .dot file produced | 209 |
| 10.6.5 | The .svg file produced | 211 |
| 10.7 | Creating K_3 with bundled edges and vertices | 212 |
| 10.7.1 | Graph | 212 |
| 10.7.2 | Function to create such a graph | 214 |
| 10.7.3 | Creating such a graph | 215 |
| 10.7.4 | The .dot file produced | 215 |
| 10.7.5 | The .svg file produced | 217 |
| 11 | Working on graphs with bundled edges and vertices | 218 |
| 11.1 | Has a my_bundled_edge | 218 |
| 11.2 | Find a my_bundled_edge | 219 |
| 11.3 | Get an edge its my_bundled_edge | 221 |
| 11.4 | Set an edge its my_bundled_edge | 222 |
| 11.5 | Storing a graph with bundled edges and vertices as a .dot | 224 |
| 11.6 | Load a directed graph with bundled edges and vertices from a .dot file | 225 |
| 11.7 | Load an undirected graph with bundled edges and vertices from a .dot file | 229 |

| | |
|--|------------|
| 12 Building graphs with custom vertices | 232 |
| 12.1 Creating the vertex class | 232 |
| 12.2 Installing the new vertex property | 234 |
| 12.3 Create the empty directed graph with custom vertices | 235 |
| 12.4 Create the empty undirected graph with custom vertices | 236 |
| 12.5 Add a custom vertex | 237 |
| 12.6 Getting the vertices' my_vertexes | 238 |
| 12.7 Creating a two-state Markov chain with custom vertices | 240 |
| 12.7.1 Graph | 240 |
| 12.7.2 Function to create such a graph | 240 |
| 12.7.3 Creating such a graph | 241 |
| 12.7.4 The .dot file produced | 242 |
| 12.7.5 The .svg file produced | 243 |
| 12.8 Creating K_2 with custom vertices | 243 |
| 12.8.1 Graph | 243 |
| 12.8.2 Function to create such a graph | 244 |
| 12.8.3 Creating such a graph | 244 |
| 12.8.4 The .dot file produced | 245 |
| 12.8.5 The .svg file produced | 246 |
| 12.9 ► Creating a path graph with custom vertices | 246 |
| 12.9.1 Graph | 246 |
| 12.9.2 Function to create such a graph | 246 |
| 12.9.3 Creating such a graph | 247 |
| 12.9.4 The .dot file produced | 248 |
| 12.9.5 The .svg file produced | 248 |
| 13 Working on graphs with custom vertices (as a custom property) | 249 |
| 13.1 Has a custom vertex with a my_vertex | 250 |
| 13.2 Find a custom vertex with a certain my_vertex | 251 |
| 13.3 Get a custom vertex its my_vertex | 253 |
| 13.4 Set a custom vertex its my_vertex | 255 |
| 13.5 Setting all custom vertices' my_vertex objects | 256 |
| 13.6 ► Are two graphs with custom vertices isomorphic? | 258 |
| 13.7 Storing a graph with custom vertices as a .dot | 261 |
| 13.8 Loading a directed graph with custom vertices from a .dot | 262 |
| 13.9 Loading an undirected graph with custom vertices from a .dot | 264 |
| 14 Building graphs with custom and selectable vertices | 266 |
| 14.1 Installing the new is_selected property | 267 |
| 14.2 Create an empty directed graph with custom and selectable vertices | 268 |
| 14.3 Create an empty undirected graph with custom and selectable vertices | 270 |
| 14.4 Add a custom and selectable vertex | 271 |
| 14.5 Creating a Markov-chain with custom and selectable vertices | 274 |
| 14.5.1 Graph | 274 |
| 14.5.2 Function to create such a graph | 274 |

| | | |
|-----------|--|------------|
| 14.5.3 | Creating such a graph | 276 |
| 14.5.4 | The .dot file produced | 277 |
| 14.5.5 | The .svg file produced | 278 |
| 14.6 | Creating K_2 with custom and selectable vertices | 279 |
| 14.6.1 | Graph | 279 |
| 14.6.2 | Function to create such a graph | 281 |
| 14.6.3 | Creating such a graph | 282 |
| 14.6.4 | The .dot file produced | 282 |
| 14.6.5 | The .svg file produced | 283 |
| 15 | Working on graphs with custom and selectable vertices | 283 |
| 15.1 | ► Getting the vertices with a certain selectedness | 284 |
| 15.2 | ► Counting the vertices with a certain selectedness | 284 |
| 15.3 | ► Adding an edge between two selected vertices | 285 |
| 15.4 | Storing a graph with custom and selectable vertices as a .dot | 287 |
| 15.5 | Loading a directed graph with custom and selectable vertices from a .dot | 291 |
| 15.6 | Loading an undirected graph with custom and selectable vertices from a .dot | 294 |
| 16 | Building graphs with custom edges and vertices | 296 |
| 16.1 | Creating the custom edge class | 297 |
| 16.2 | Installing the new edge property | 299 |
| 16.3 | Create an empty directed graph with custom edges and vertices | 300 |
| 16.4 | Create an empty undirected graph with custom edges and vertices | 302 |
| 16.5 | Add a custom edge | 303 |
| 16.6 | Getting the custom edges my_edges | 305 |
| 16.7 | Creating a Markov-chain with custom edges and vertices | 306 |
| 16.7.1 | Graph | 306 |
| 16.7.2 | Function to create such a graph | 307 |
| 16.7.3 | Creating such a graph | 309 |
| 16.7.4 | The .dot file produced | 310 |
| 16.7.5 | The .svg file produced | 310 |
| 16.8 | Creating K_3 with custom edges and vertices | 310 |
| 16.8.1 | Graph | 310 |
| 16.8.2 | Function to create such a graph | 312 |
| 16.8.3 | Creating such a graph | 313 |
| 16.8.4 | The .dot file produced | 313 |
| 16.8.5 | The .svg file produced | 314 |
| 17 | Working on graphs with custom edges and vertices | 314 |
| 17.1 | Has a my_custom_edge | 314 |
| 17.2 | Find a my_custom_edge | 316 |
| 17.3 | Get an edge its my_custom_edge | 318 |
| 17.4 | Set an edge its my_custom_edge | 319 |
| 17.5 | ► Counting the edges with a certain selectedness | 321 |

| | | |
|-----------|--|------------|
| 17.6 | Storing a graph with custom edges and vertices as a .dot | 323 |
| 17.7 | Load a directed graph with custom edges and vertices from a .dot file | 324 |
| 17.8 | Load an undirected graph with custom edges and vertices from a .dot file | 327 |
| 18 | Building graphs with custom and selectable edges and vertices | 329 |
| 18.1 | Installing the new is_selected property | 330 |
| 18.2 | Create an empty directed graph with custom and selectable edges and vertices | 331 |
| 18.3 | Create an empty undirected graph with custom and selectable edges and vertices | 333 |
| 18.4 | Add a custom and selectable edge | 334 |
| 18.5 | Creating a Markov-chain with custom and selectable vertices . . | 336 |
| 18.5.1 | Graph | 336 |
| 18.5.2 | Function to create such a graph | 337 |
| 18.5.3 | Creating such a graph | 339 |
| 18.5.4 | The .dot file produced | 340 |
| 18.5.5 | The .svg file produced | 341 |
| 18.6 | Creating K_2 with custom and selectable edges and vertices . . . | 342 |
| 18.6.1 | Graph | 342 |
| 18.6.2 | Function to create such a graph | 344 |
| 18.6.3 | Creating such a graph | 345 |
| 18.6.4 | The .dot file produced | 345 |
| 18.6.5 | The .svg file produced | 346 |
| 19 | Working on graphs with custom and selectable edges and ver- tices | 346 |
| 19.1 | Storing a graph with custom and selectable edges and vertices as a .dot | 347 |
| 19.2 | Loading a directed graph with custom and selectable edges and vertices from a .dot | 348 |
| 19.3 | Loading an undirected graph with custom and selectable edges and vertices from a .dot | 351 |
| 20 | Building graphs with a graph name | 354 |
| 20.1 | Create an empty directed graph with a graph name property . . | 354 |
| 20.2 | Create an empty undirected graph with a graph name property . | 355 |
| 20.3 | Get a graph its name property | 357 |
| 20.4 | Set a graph its name property | 358 |
| 20.5 | Create a directed graph with a graph name property | 358 |
| 20.5.1 | Graph | 358 |
| 20.5.2 | Function to create such a graph | 359 |
| 20.5.3 | Creating such a graph | 359 |
| 20.5.4 | The .dot file produced | 360 |
| 20.5.5 | The .svg file produced | 361 |

| | | |
|-----------|---|------------|
| 20.6 | Create an undirected graph with a graph name property | 361 |
| 20.6.1 | Graph | 361 |
| 20.6.2 | Function to create such a graph | 361 |
| 20.6.3 | Creating such a graph | 362 |
| 20.6.4 | The .dot file produced | 363 |
| 20.6.5 | The .svg file produced | 363 |
| 21 | Working on graphs with a graph name | 364 |
| 21.1 | Storing a graph with a graph name property as a .dot file | 364 |
| 21.2 | Loading a directed graph with a graph name property from a .dot file | 364 |
| 21.3 | Loading an undirected graph with a graph name property from a .dot file | 366 |
| 22 | Other graph functions | 368 |
| 22.1 | Encode a std::string to a Graphviz-friendly format | 368 |
| 22.2 | Decode a std::string from a Graphviz-friendly format | 368 |
| 22.3 | Check if a std::string is Graphviz-friendly | 368 |
| 23 | Misc functions | 369 |
| 23.1 | Getting a data type as a std::string | 369 |
| 23.2 | Convert a .dot to .svg | 370 |
| 23.3 | Check if a file exists | 371 |
| 24 | Errors | 372 |
| 24.1 | Formed reference to void | 372 |
| 24.2 | No matching function for call to ‘clear_out_edges’ | 372 |
| 24.3 | No matching function for call to ‘clear_in_edges’ | 373 |
| 24.4 | Undefined reference to boost::detail::graph::read_graphviz_new . | 373 |
| 24.5 | Property not found: node_id | 373 |
| 24.6 | Stream zeroes | 374 |
| 25 | Appendix | 376 |
| 25.1 | List of all edge, graph and vertex properties | 376 |
| 25.2 | Graphviz attributes | 376 |

1 Introduction

This is ‘A well-connected C++11 Boost.Graph tutorial’, version 1.10.

1.1 Why this tutorial

I needed this tutorial already in 2006, when I started experimenting with Boost.Graph. More specifically, I needed a tutorial that:

- Orders concepts chronologically

- Increases complexity gradually
- Shows complete pieces of code

What I had were the book [8] and the Boost.Graph website, both did not satisfy these requirements.

1.2 Tutorial style

This tutorial is aimed at the beginner programmer. This tutorial is intended to take the reader to the level of understanding the book [8] and the Boost.Graph website require. It is about basic graph manipulation, not the more advanced graph algorithms.

This tutorial is intended to be as verbose, such that a beginner should be able to follow every step, from reading the tutorial from beginning to end chronologically. Especially in the earlier chapters, the rationale behind the code presented is given, including references to the literature. Chapters marked with '►' are optional, less verbose and bring no new information to the storyline.

This tutorial is intended to be as repetitive, such that a beginner can spot the patterns in the code snippets their increasing complexity. Extending code from this tutorial should be as easy as extending the patterns.

In the index, I did first put all my long-named functions there literally, but this resulted in a very sloppy layout. Instead, the function 'do_something' can be found as 'Do something' in the index. On the other hand, STL and Boost functions like 'std::do_something' and 'boost::do_something' can be found as such in the index.

1.3 Code snippets

For every concept, I will show

- the 'do' function: a function that achieves a goal, for example 'create_empty_undirected_graph'
- the 'demo' function: a function that demonstrates how to call the first, for example 'create_empty_undirected_graph_demo'

I enjoy to show concepts by putting those in (long-named) functions. These functions sometimes border the trivial, by, for example, only calling a single Boost.Graph function. On the other hand, these functions have more English-sounding names, resulting in demonstration code that is readable. Additionally, they explicitly mention their return type (in a simpler way), which may be considered informative.

All coding snippets are taken from compiled C++11 code. I chose to use C++11 because (1) C++14 was not installable on all my computers (2) the step to C++14 is small. All code is tested to compile cleanly under GCC at the highest warning level. The code, as well as this tutorial, can be downloaded from the GitHub at www.github.com/richeibilderbeek/BoostGraphTutorial.

1.4 Coding style

I use the coding style from the Core C++ Guidelines. At the time of this writing, the Core C++ Guidelines were still in early development, so I can only hope the conventions I then chose to follow are still Good Ideas.

It is important to add comments to code. In this tutorial, however, I have chosen not to put comments in code, as I already describe the function in the tutorial its text. This way, it prevents me from saying the same things twice.

It is good to write generic code. In this tutorial, however, I have chosen my functions to have no templated arguments for conciseness and readability. For example, a vertex name is `std::string`, the type for if a vertex is selected is a boolean, and the custom vertex type is of type `'my_custom_vertex'`. I think these choices are reasonable and that the resulting increase in readability is worth it.

Due to my long function names and the limitation of ≈ 50 characters per line, sometimes the code does get to look a bit awkward. I am sorry for this.

I prefer to use the keyword `auto` over doubling the lines of code for using statements. Because the `'do'` functions return an explicit data type, these can be used for reference (until `'decltype(auto)'` gets into fashion as a return type). If you really want to know a type, you can use the `'get_type_name'` function (chapter 23.1).

On the other hand, I am explicit in the namespaces of functions and classes I use, so to distinguish between types like `'std::array'` and `'boost::array'`. Some functions (for example, `'get'`) reside in the namespace of the graph to work on. In this tutorial, this is in the global namespace. Thus, I will write `'get'`, instead of `'boost::get'`, as the latter does not compile.

I try to use STL algorithms wherever I can. Also you should prefer algorithm calls over hand-written for-loops ([9] chapter 18.12.1, [7] item 43). Sometimes using these algorithms becomes a burden on the lines of code. This is because in C++11, a lambda function argument (use by the algorithm) must have its data type specified. It may take multiple lines of `'using'` statements being able to do so. In C++14 one can use `'auto'` there as well. So, only if it shortens the number of lines significantly, I use raw for-loops, even though you shouldn't.

1.5 License

This tutorial is licensed under Creative Commons license 4.0. All C++ code is licensed under GPL 3.0.



Figure 1: Creative Commons license 4.0

1.6 Feedback

This tutorial is not intended to be perfect yet. For that, I need help and feedback from the community. All referenced feedback is welcome, as well as any constructive feedback.

I have tried hard to strictly follow the style as described above. If you find I deviated from these decisions somewhere, I would be grateful if you'd let know. Next to this, there are some sections that need to be coded or have its code improved.

1.7 Acknowledgements

These are users that improved this tutorial and/or the code behind this tutorial, in chronological order:

- m-dudley, <http://stackoverflow.com/users/111327/m-dudley>
- E. Kawashima
- mat69, <https://www.reddit.com/user/mat69>
- danielhj, <https://www.reddit.com/user/danielhj>
- sehe, <http://stackoverflow.com/users/85371/sehe>
- cv_and_me, <http://stackoverflow.com/users/2417774/cv-and-he>

1.8 Outline

The chapters of this tutorial are also like a well-connected graph (as shown in figure 2). To allow for quicker learners to skim chapters, or for beginners looking to find the patterns.

The distinction between the chapter is in the type of edges and vertices. They can have:

- no properties: see chapter 2
- have a name: see chapter 4
- have a bundled property: see chapter 8
- have a custom property: see chapter 12

The differences between graphs with bundled and custom properties are shown in table 1:

| | Bundled | Custom |
|----------------------|-----------------------------------|---|
| Meaning | Edges/vertices are of your type | Edges/vertices have an additional custom property |
| Interface | Directly | Via property map |
| Class members | Must be public | Can be private |
| File I/O mechanism | Via public class members | Via stream operators |
| File I/O constraints | Restricted to Graphviz attributes | Need encoding and decoding |

Table 1: Difference between bundled and custom properties

Pivotal chapters are chapters like ‘Finding the first vertex with ...’, as this opens up the door to finding a vertex and manipulating it.

All chapters have a rather similar structure in themselves, as depicted in figure 3.

There are also some bonus chapters, that I have labeled with a ‘►’. These chapters are added I needed these functions myself and adding them would not hurt. Just feel free to skip them, as there will be less theory explained.

2 Building graphs without properties

Boost.Graph is about creating graphs. In this chapter we create the simplest of graphs, in which edges and nodes have no properties (e.g. having a name).

Still, there are two types of graphs that can be constructed: undirected and directed graphs. The difference between directed and undirected graphs is in the edges: in an undirected graph, an edge connects two vertices without any directionality, as displayed in figure 4. In a directed graph, an edge goes from a certain vertex, its source, to another (which may actually be the same), its target. A directed graph is shown in figure 5.

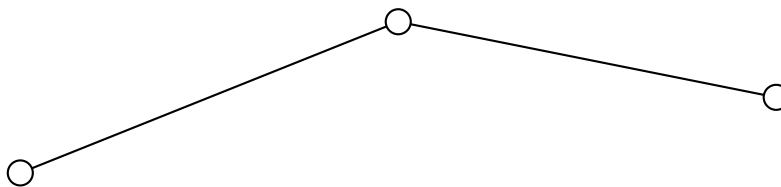


Figure 4: Example of an undirected graph

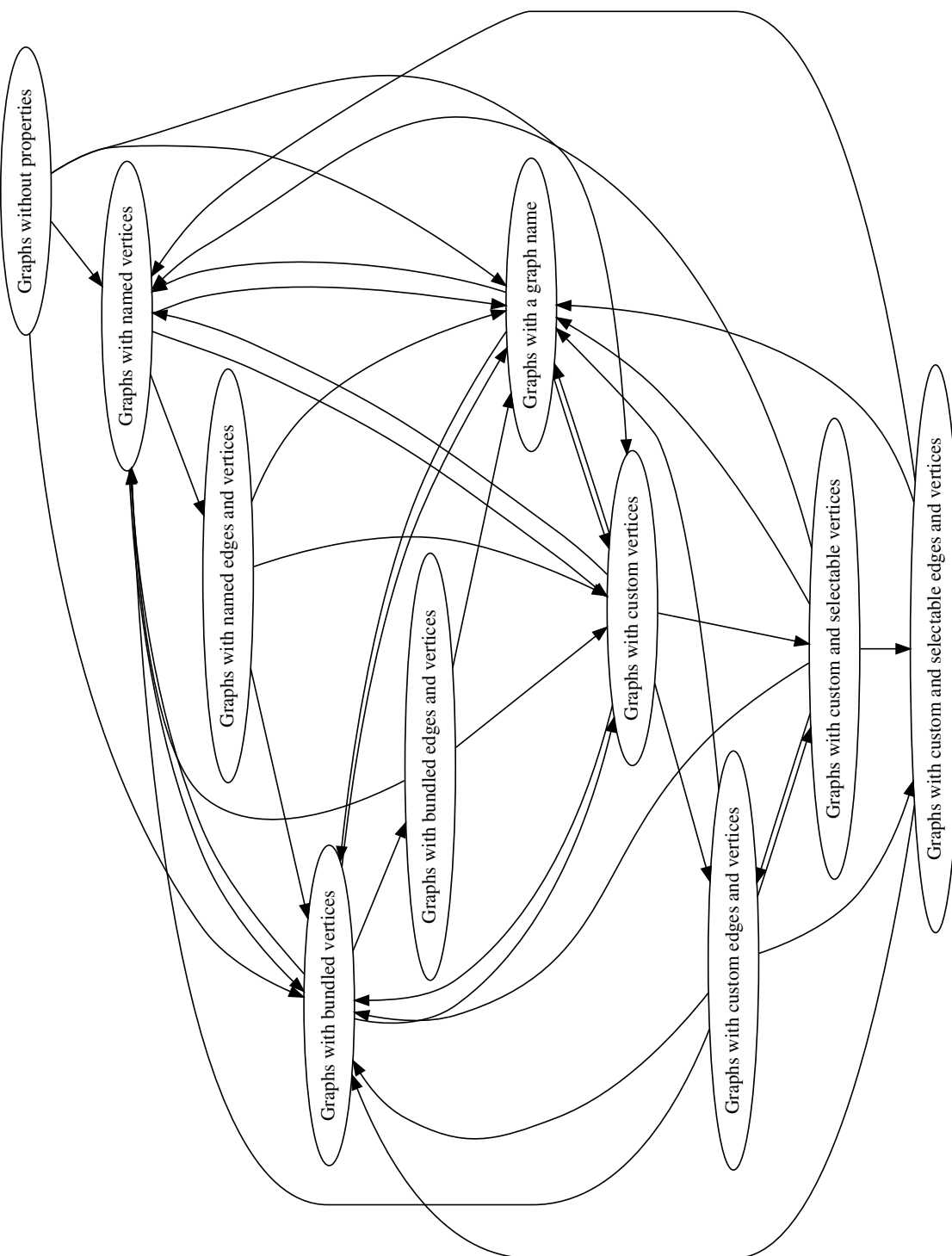


Figure 2: The relations between chapters

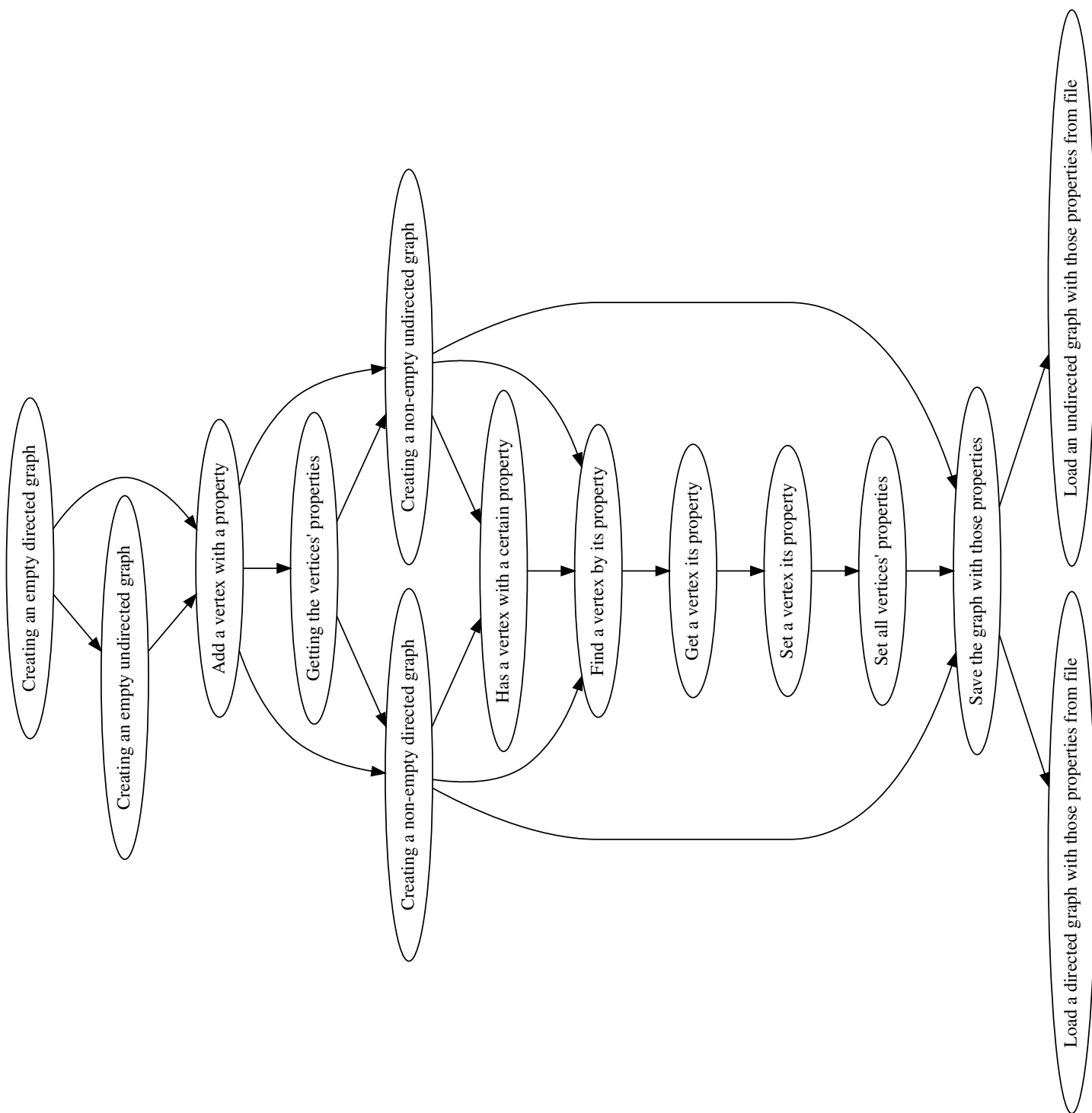


Figure 3: The relations between sub-chapters

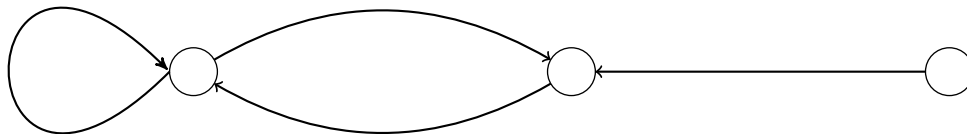


Figure 5: Example of a directed graph

In this chapter, we will build two directed and two undirected graphs:

- An empty (directed) graph, which is the default type: see chapter 2.1
- An empty (undirected) graph: see chapter 2.2
- A two-state Markov chain, a directed graph with two vertices and four edges, chapter 2.14
- K_2 , an undirected graph with two vertices and one edge, chapter 2.15

Creating an empty graph may sound trivial, it is not, thanks to the versatility of the Boost.Graph library.

In the process of creating graphs, some basic (sometimes bordering trivial) functions are encountered:

- Counting the number of vertices: see chapter 2.3
- Counting the number of edges: see chapter 2.4
- Adding a vertex: see chapter 2.5
- Getting all vertices: see chapter 2.7
- Getting all vertex descriptors: see chapter 2.8
- Adding an edge: see chapter 2.9
- Getting all edges: see chapter 2.11
- Getting all edge descriptors: see chapter 2.13

These functions are mostly there for completion and showing which data types are used.

The chapter also introduces some important concepts:

- Vertex descriptors: see chapter 2.6
- Edge insertion result: see chapter 2.10
- Edge descriptors: see chapter 2.12

After this chapter you may want to:

- Building graphs with named vertices: see chapter 4
- Building graphs with bundled vertices: see chapter 8
- Building graphs with custom vertices: see chapter 12
- Building graphs with a graph name: see chapter 20

2.1 Creating an empty (directed) graph

Let's create an empty graph!

Algorithm 1 shows the function to create an empty graph.

Algorithm 1 Creating an empty (directed) graph

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

boost::adjacency_list<>
create_empty_directed_graph() noexcept
{
    return {};
}
```

The code consists out of an `#include` and a function definition. The `#include` tells the compiler to read the header file `'adjacency_list.hpp'`. A header file (often with a `'.h'` or `'.hpp'` extension) contains class and functions declarations and/or definitions. The header file `'adjacency_list.hpp'` contains the `boost::adjacency_list` class definition. Without including this file, you will get compile errors like `'definition of boost::adjacency_list unknown'`¹. The function `'create_empty_directed_graph'` has:

- a return type: The return type is `'boost::adjacency_list<>'`, that is a `'boost::adjacency_list'` with all template arguments set at their defaults
- a `noexcept` specification: the function should not throw², so it is preferred to mark it `noexcept` ([10] chapter 13.7).

¹In practice, these compiler error messages will be longer, bordering the unreadable

²if the function would throw because it cannot allocate this little piece of memory, you are already in big trouble

- a function body: all the function body does is implicitly create its return type by using the ‘{}’. An alternative syntax would be ‘return boost::adjacency_list<>()’, which is needlessly longer

Algorithm 2 demonstrates the ‘create_empty_directed_graph’ function. Note that it includes a header file with the same name as the function³ first, to be able to use it. ‘auto’ is used, as this is preferred over explicit type declarations ([10] chapter 31.6). The keyword ‘auto’ lets the compiler figure out the type itself.

Algorithm 2 Demonstration of ‘create_empty_directed_graph’

```
#include "create_empty_directed_graph.h"

void create_empty_directed_graph_demo() noexcept
{
    const auto g = create_empty_directed_graph();
}
```

Congratulations, you’ve just created a boost::adjacency_list with its default template arguments. The boost::adjacency_list is the most commonly used graph type, the other is the boost::adjacency_matrix. We do not do anything with it yet, but still, you’ve just created a graph, in which:

- The out edges and vertices are stored in a std::vector
- The edges have a direction
- The vertices, edges and graph have no properties
- The edges are stored in a std::list

It stores its edges, out edges and vertices in a two different STL⁴ containers. std::vector is the container you should use by default ([10] chapter 31.6, [11] chapter 76), as it has constant time look-up and back insertion. The std::list is used for storing the edges, as it is better suited at inserting elements at any position.

I use const to store the empty graph as we do not modify it. Correct use of const is called const-correct. Prefer to be const-correct ([9] chapter 7.9.3, [10] chapter 12.7, [7] item 3, [3] chapter 3, [11] item 15, [2] FAQ 14.05, [1] item 8, [4] 9.1.6).

2.2 Creating an empty undirected graph

Let’s create another empty graph! This time, we even make it undirected!

³I do not think it is important to have creative names

⁴Standard Template Library, the standard library

Algorithm 3 shows how to create an undirected graph.

Algorithm 3 Creating an empty undirected graph

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

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS
>
create_empty_undirected_graph() noexcept
{
    return {};
}
```

This algorithm differs from the ‘create_empty_directed_graph’ function (algorithm 1) in that there are three template arguments that need to be specified in the creation of the `boost::adjacency_list`:

- the first ‘`boost::vecS`’: select (that is what the ‘S’ means) that out edges are stored in a `std::vector`. This is the default way.
- the second ‘`boost::vecS`’: select that the graph vertices are stored in a `std::vector`. This is the default way.
- ‘`boost::undirectedS`’: select that the graph is undirected. This is all we needed to change. By default, this argument is `boost::directed`

Algorithm 4 demonstrates the ‘create_empty_undirected_graph’ function.

Algorithm 4 Demonstration of ‘create_empty_undirected_graph’

```
#include "create_empty_undirected_graph.h"

void create_empty_undirected_graph_demo() noexcept
{
    const auto g = create_empty_undirected_graph();
}
```

Congratulations, with algorithm 4, you’ve just created an undirected graph in which:

- The out edges and vertices are stored in a `std::vector`
- The graph is undirected

- Vertices, edges and graph have no properties
- Edges are stored in a `std::list`

2.3 Counting the number of vertices

Let's count all zero vertices of an empty graph!

Algorithm 5 Count the number of vertices

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

template <typename graph>
int get_n_vertices(const graph& g) noexcept
{
    const int n{
        static_cast<int>(boost::num_vertices(g))
    };
    assert(static_cast<unsigned long>(n)
        == boost::num_vertices(g)
    );
    return n;
}
```

The function ‘`get_n_vertices`’ takes the result of `boost::num_vertices`, converts it to `int` and checks if there was conversion error. We do so, as one should prefer using signed data types over unsigned ones in an interface ([4] chapter 9.2.2). To do so, in the function body its first statement, the unsigned long produced by `boost::num_vertices` get converted to an `int` using a `static_cast`. Using an unsigned integer over a (signed) integer for the sake of gaining that one more bit ([9] chapter 4.4) should be avoided. The integer ‘`n`’ is initialized using list-initialization, which is preferred over the other initialization syntaxes ([10] chapter 17.7.6).

The `assert` checks if the conversion back to unsigned long re-creates the original value, to check if no information has been lost. If information is lost, the program crashes. Use `assert` extensively ([9] chapter 24.5.18, [10] chapter 30.5, [11] chapter 68, [6] chapter 8.2, [5] hour 24, [4] chapter 2.6).

The function ‘`get_n_vertices`’ is demonstrated in algorithm 6, to measure the number of vertices of both the directed and undirected graph we are already able to create.

Algorithm 6 Demonstration of the ‘get_n_vertices’ function

```
#include <cassert>
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"
#include "get_n_vertices.h"

void get_n_vertices_demo() noexcept
{
    const auto g = create_empty_directed_graph();
    assert(get_n_vertices(g) == 0);

    const auto h = create_empty_undirected_graph();
    assert(get_n_vertices(h) == 0);
}
```

Note that the type of graph does not matter here. One can count the number of vertices of every graph, as all graphs have vertices. Boost.Graph is very good at detecting operations that are not allowed, during compile time.

2.4 Counting the number of edges

Let’s count all zero edges of an empty graph!

This is very similar to the previous chapter, only it uses `boost::num_edges` instead:

Algorithm 7 Count the number of edges

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

template <typename graph>
int get_n_edges(const graph& g) noexcept
{
    const int n{
        static_cast<int>(boost::num_edges(g))
    };
    assert(static_cast<unsigned long>(n)
        == boost::num_edges(g)
    );
    return n;
}
```

This code is similar to the ‘get_n_vertices’ function (algorithm 5, see rationale there) except ‘`boost::num_edges`’ is used, instead of ‘`boost::num_vertices`’,

which also returns an unsigned long.

The function ‘get_n_edges’ is demonstrated in algorithm 8, to measure the number of edges of an empty directed and undirected graph.

Algorithm 8 Demonstration of the ‘get_n_edges’ function

```
#include <cassert>

#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"
#include "get_n_edges.h"

void get_n_edges_demo() noexcept
{
    const auto g = create_empty_directed_graph();
    assert(get_n_edges(g) == 0);

    const auto h = create_empty_undirected_graph();
    assert(get_n_edges(h) == 0);
}
```

2.5 Adding a vertex

Empty graphs are nice, now its time to add a vertex!

To add a vertex to a graph, the boost::add_vertex function is used as shows in algorithm 9:

Algorithm 9 Adding a vertex to a graph

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

template <typename graph>
typename boost::graph_traits<graph>::vertex_descriptor
add_vertex(graph& g) noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const");
    const auto vd = boost::add_vertex(g);
    return vd;
}
```

The static_assert at the top of the function checks during compiling if the

function is called with a non-const graph. One can freely omit this `static_assert`: you will get a compiler error anyways, be it a less helpful one.

Note that `boost::add_vertex` (in the ‘`add_vertex`’ function) returns a vertex descriptor, which is ignored for now. Vertex descriptors are looked at in more details at the chapter 2.6, as we need these to add an edge. To allow for this already, ‘`add_vertex`’ also returns a vertex descriptor.

Algorithm 10 shows how to add a vertex to a directed and undirected graph.

Algorithm 10 Demonstration of the ‘`add_vertex`’ function

```
#include "add_vertex.h"
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"

void add_vertex_demo() noexcept
{
    auto g = create_empty_undirected_graph();
    add_vertex(g);
    assert(boost::num_vertices(g) == 1);

    auto h = create_empty_directed_graph();
    add_vertex(h);
    assert(boost::num_vertices(h) == 1);
}
```

This demonstration code creates two empty graphs, adds one vertex to each and then asserts that the number of vertices in each graph is one. This works for both types of graphs, as all graphs have vertices.

2.6 Vertex descriptors

A vertex descriptor is a handle to a vertex within a graph.

Vertex descriptors can be obtained by dereferencing a vertex iterator (see chapter 2.8). To do so, we first obtain some vertex iterators in chapter 2.7).

Vertex descriptors are used to:

- add and edge between two vertices, see chapter 2.9
- obtain properties of vertex a vertex, for example the vertex its out degrees (chapter 3.1), the vertex its name (chapter 4.4), or a custom vertex property (chapter 12.6)

In this tutorial, vertex descriptors have named prefixed with ‘`vd_`’, for example ‘`vd_1`’.

2.7 Get the vertex iterators

You cannot get the vertices. This may sound unexpected, as it must be possible to work on the vertices of a graph. Working on the vertices of a graph is done through these steps:

- Obtain a vertex iterator pair from the graph
- Dereferencing a vertex iterator to obtain a vertex descriptor

‘vertices’ (not ‘boost::vertices’) is used to obtain a vertex iterator pair, as shown in algorithm 11. The first vertex iterator points to the first vertex (its descriptor, to be precise), the second points to beyond the last vertex (its descriptor, to be precise). In this tutorial, vertex iterator pairs have names prefixed with ‘vip_’, for example ‘vip_1’.

Algorithm 11 Get the vertex iterators of a graph

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

template <typename graph>
std::pair<
    typename graph::vertex_iterator,
    typename graph::vertex_iterator
>
get_vertex_iterators(const graph& g) noexcept
{
    return vertices(g);
}
```

This is a somewhat trivial function, as it forwards the function call to ‘vertices’ (not ‘boost::vertices’).

These vertex iterators can be dereferenced to obtain the vertex descriptors. Note that ‘get_vertex_iterators’ will not be used often in isolation: usually one obtains the vertex descriptors immediately. Just for your reference, algorithm 12 demonstrates of the ‘get_vertices’ function, by showing that the vertex iterators of an empty graph point to the same location.

Algorithm 12 Demonstration of ‘get_vertex_iterators’

```
#include <cassert>
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"
#include "get_vertex_iterators.h"

void get_vertex_iterators_demo() noexcept
{
    const auto g = create_empty_undirected_graph();
    const auto vip_g = get_vertex_iterators(g);
    assert(vip_g.first == vip_g.second);

    const auto h = create_empty_directed_graph();
    const auto vip_h = get_vertex_iterators(h);
    assert(vip_h.first == vip_h.second);
}
```

2.8 Get all vertex descriptors

Vertex descriptors are the way to manipulate those vertices. Let’s go get the all!

Vertex descriptors are obtained from dereferencing vertex iterators. Algorithm 13 shows how to obtain all vertex descriptors from a graph.

Algorithm 13 Get all vertex descriptors of a graph

```
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/graph_traits.hpp>

template <typename graph>
std::vector<
    typename boost::graph_traits<graph>::vertex_descriptor
>
get_vertex_descriptors(const graph& g) noexcept
{
    using vd = typename graph::vertex_descriptor;

    std::vector<vd> vds(boost::num_vertices(g));
    const auto vis = vertices(g);
    std::copy(vis.first, vis.second, std::begin(vds));
    return vds;
}
```

This is the first more complex piece of code. In the first lines, some ‘using’ statements allow for shorter type names⁵.

The `std::vector` to serve as a return value is created at the needed size, which is the number of vertices.

The function ‘vertices’ (not `boost::vertices`!) returns a vertex iterator pair. These iterators are used by `std::copy` to iterate over. `std::copy` is an STL algorithm to copy a half-open range. Prefer algorithm calls over hand-written for-loops ([9] chapter 18.12.1, [7] item 43).

In this case, we copy all vertex descriptors in the range produced by ‘vertices’ to the `std::vector`.

This function will not be used in practice: one iterates over the vertices directly instead, saving the cost of creating a `std::vector`. This function is only shown as an illustration.

Algorithm 14 demonstrates that an empty graph has no vertex descriptors:

Algorithm 14 Demonstration of ‘get_vertex_descriptors’

```
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"
#include "get_vertex_descriptors.h"

void get_vertex_descriptors_demo() noexcept
{
    const auto g = create_empty_undirected_graph();
    const auto vds_g = get_vertex_descriptors(g);
    assert(vds_g.empty());

    const auto h = create_empty_directed_graph();
    const auto vds_h = get_vertex_descriptors(h);
    assert(vds_h.empty());
}
```

Because all graphs have vertices and thus vertex descriptors, the type of graph is unimportant for this code to compile.

2.9 Add an edge

To add an edge to a graph, two vertex descriptors are needed. A vertex descriptor is a handle to the vertex within a graph (vertex descriptors are looked at in more details in chapter 2.6). Algorithm 15 adds two vertices to a graph, and connects these two using `boost::add_edge`:

⁵which may be necessary just to create a tutorial with code snippets that are readable

Algorithm 15 Adding (two vertices and) an edge to a graph

```
#include <cassert>
#include <type_traits>
#include <boost/graph/adjacency_list.hpp>

template <typename graph>
typename boost::graph_traits<graph>::edge_descriptor
add_edge(graph& g) noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const");
    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);
    const auto aer = boost::add_edge(
        vd_a, vd_b, g
    );
    assert(aer.second);
    return aer.first;
}
```

Algorithm 15 shows how to add an isolated edge to a graph (instead of allowing for graphs with higher connectivities). First, two vertices are created, using the function ‘boost::add_vertex’. ‘boost::add_vertex’ returns a vertex descriptor (which I prefix with ‘vd’), both of which are stored. The vertex descriptors are used to add an edge to the graph, using ‘boost::add_edge’. ‘boost::add_edge’ returns a std::pair, consisting of an edge descriptor and a boolean success indicator. The success of adding the edge is checked by an assert statement. Here we assert that this insertion was successful. Insertion can fail if an edge is already present and duplicates are not allowed.

A demonstration of add_edge is shown in algorithm 16, in which an edge is added to both a directed and undirected graph, after which the number of edges and vertices is checked.

Algorithm 16 Demonstration of ‘add_edge’

```
#include "add_edge.h"
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"

void add_edge_demo() noexcept
{
    auto g = create_empty_undirected_graph();
    add_edge(g);
    assert(boost::num_vertices(g) == 2);
    assert(boost::num_edges(g) == 1);

    auto h = create_empty_directed_graph();
    add_edge(h);
    assert(boost::num_vertices(h) == 2);
    assert(boost::num_edges(h) == 1);
}
```

The graph type is unimportant: as all graph types have vertices and edges, edges can be added without possible compile problems.

2.10 boost::add_edge result

When using the function ‘boost::add_edge’, a ‘std::pair<edge_descriptor, bool>’ is returned. It contains both the edge descriptor (see chapter 2.12) and a boolean, which indicates insertion success.

In this tutorial, boost::add_edge results have named prefixed with ‘aer_’, for example ‘aer_1’.

2.11 Getting the edge iterators

You cannot get the edges directly. Instead, working on the edges of a graph is done through these steps:

- Obtain an edge iterator pair from the graph
- Dereference an edge iterator to obtain an edge descriptor

‘edges’ (not boost::edges!) is used to obtain an edge iterator pair. The first edge iterator points to the first edge (its descriptor, to be precise), the second points to beyond the last edge (its descriptor, to be precise). In this tutorial, edge iterator pairs have named prefixed with ‘eip_’, for example ‘eip_1’. Algorithm 17 shows how to obtain these:

Algorithm 17 Get the edge iterators of a graph

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

template <typename graph>
std::pair<
    typename graph::edge_iterator,
    typename graph::edge_iterator
>
get_edge_iterators(const graph& g) noexcept
{
    return edges(g);
}
```

This is a somewhat trivial function, as all it does is forward to function call to ‘edges’ (not `boost::edges`!) These edge iterators can be dereferenced to obtain the edge descriptors. Note that this function will not be used often in isolation: usually one obtains the edge descriptors immediately.

Algorithm 18 demonstrates ‘get_edge_iterators’ by showing that both iterators of the edge iterator pair point to the same location, when the graph is empty.

Algorithm 18 Demonstration of ‘get_edge_iterators’

```
#include <cassert>
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"
#include "get_edge_iterators.h"

void get_edge_iterators_demo() noexcept
{
    const auto g = create_empty_undirected_graph();
    const auto eip_g = get_edge_iterators(g);
    assert(eip_g.first == eip_g.second);

    auto h = create_empty_directed_graph();
    const auto eip_h = get_edge_iterators(h);
    assert(eip_h.first == eip_h.second);
}
```

2.12 Edge descriptors

An edge descriptor is a handle to an edge within a graph. They are similar to vertex descriptors (chapter 2.6).

Edge descriptors are used to obtain the name, or other properties, of an edge. In this tutorial, edge descriptors have names prefixed with ‘ed_’, for example ‘ed_1’.

2.13 Get all edge descriptors

Obtaining all edge descriptors is similar to obtaining all vertex descriptors (algorithm 13), as shown in algorithm 19:

Algorithm 19 Get all edge descriptors of a graph

```
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include "boost/graph/graph_traits.hpp"

template <typename graph>
std::vector<
    typename boost::graph_traits<graph>::edge_descriptor
> get_edge_descriptors(const graph& g) noexcept
{
    using boost::graph_traits;
    using ed = typename graph_traits<graph>::
        edge_descriptor;
    std::vector<ed> v(boost::num_edges(g));
    const auto eip = edges(g);
    std::copy(eip.first, eip.second, std::begin(v));
    return v;
}
```

The only difference is that instead of the function ‘vertices’ (not `boost::vertices`!), ‘edges’ (not `boost::edges`!) is used.

Algorithm 20 demonstrates the ‘get_edge_descriptor’, by showing that empty graphs do not have any edge descriptors.

Algorithm 20 Demonstration of `get_edge_descriptors`

```
#include <cassert>
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"
#include "get_edge_descriptors.h"

void get_edge_descriptors_demo() noexcept
{
    const auto g = create_empty_directed_graph();
    const auto eds_g = get_edge_descriptors(g);
    assert(eds_g.empty());

    const auto h = create_empty_undirected_graph();
    const auto eds_h = get_edge_descriptors(h);
    assert(eds_h.empty());
}
```

2.14 Creating a directed graph

Finally, we are going to create a directed non-empty graph!

2.14.1 Graph

This directed graph is a two-state Markov chain, with two vertices and four edges, as depicted in figure 6:

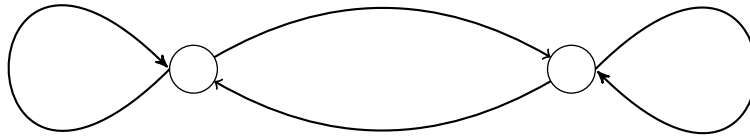


Figure 6: The two-state Markov chain

Note that directed graphs can have edges that start and end in the same vertex. These are called self-loops.

2.14.2 Function to create such a graph

To create this two-state Markov chain, the following code can be used:

Algorithm 21 Creating the two-state Markov chain as depicted in figure 6

```
#include <cassert>
#include "create_empty_directed_graph.h"

boost::adjacency_list<>
create_markov_chain() noexcept
{
    auto g = create_empty_directed_graph();
    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);
    const auto aer_aa = boost::add_edge(vd_a, vd_a, g);
    assert(aer_aa.second);
    const auto aer_ab = boost::add_edge(vd_a, vd_b, g);
    assert(aer_ab.second);
    const auto aer_ba = boost::add_edge(vd_b, vd_a, g);
    assert(aer_ba.second);
    const auto aer_bb = boost::add_edge(vd_b, vd_b, g);
    assert(aer_bb.second);
    return g;
}
```

Instead of typing the complete type, we call the ‘create_empty_directed_graph’ function, and let auto figure out the type. The vertex descriptors (see chapter 2.6) created by two boost::add_vertex calls are stored to add an edge to the graph. Then boost::add_edge is called four times. Every time, its return type (see chapter 2.10) is checked for a successful insertion.

Note that the graph lacks all properties: nodes do not have names, nor do edges.

2.14.3 Creating such a graph

Algorithm 22 demonstrates the ‘create_markov_chain_graph’ function and checks if it has the correct amount of edges and vertices:

Algorithm 22 Demonstration of the ‘create_markov_chain’

```
#include <cassert>
#include "create_markov_chain.h"

void create_markov_chain_demo() noexcept
{
    const auto g = create_markov_chain();
    assert(boost::num_vertices(g) == 2);
    assert(boost::num_edges(g) == 4);
}
```

2.14.4 The .dot file produced

Running a bit ahead, this graph can be converted to a .dot file using the ‘save_graph_to_dot’ function (algorithm 48). The .dot file created is displayed in algorithm 23:

Algorithm 23 .dot file created from the ‘create_markov_chain_graph’ function (algorithm 21), converted from graph to .dot file using algorithm 48

```
digraph G {
0;
1;
0->0 ;
0->1 ;
1->0 ;
1->1 ;
}
```

From the .dot file one can already see that the graph is directed, because:

- The first word, ‘digraph’, denotes a directed graph (where ‘graph’ would have indicated an undirectional graph)
- The edges are written as ‘->’ (where undirected connections would be written as ‘-’)

2.14.5 The .svg file produced

The .svg file of this graph is shown in figure 7:

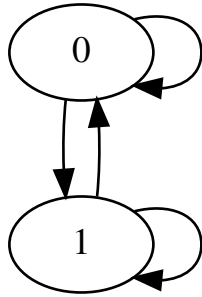


Figure 7: .svg file created from the ‘create_markov_chain’ function (algorithm 21) its .dot file and converted from .dot file to .svg using algorithm 339

This figure shows that the graph is directed, as the edges have arrow heads. The vertices display the node index, which is the default behavior.

2.15 Creating K_2 , a fully connected undirected graph with two vertices

Finally, we are going to create an undirected non-empty graph!

2.15.1 Graph

To create a fully connected undirected graph with two vertices (also called K_2), one needs two vertices and one (undirected) edge, as depicted in figure 8.



Figure 8: K_2 : a fully connected undirected graph with two vertices

2.15.2 Function to create such a graph

To create K_2 , the following code can be used:

Algorithm 24 Creating K_2 as depicted in figure 8

```
#include "create_empty_undirected_graph.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS
>
create_k2_graph() noexcept
{
    auto g = create_empty_undirected_graph();
    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);
    const auto aer = boost::add_edge(vd_a, vd_b, g);
    assert(aer.second);
    return g;
}
```

This code is very similar to the ‘add_edge’ function (algorithm 15). Instead of typing the graph its type, we call the ‘create_empty_undirected_graph’ function and let auto figure it out. The vertex descriptors (see chapter 2.6) created by two boost::add_vertex calls are stored to add an edge to the graph. From boost::add_edge its return type (see chapter 2.10), it is only checked that insertion has been successful.

Note that the graph lacks all properties: nodes do not have names, nor do edges.

2.15.3 Creating such a graph

Algorithm 25 demonstrates how to ‘create_k2_graph’ and checks if it has the correct amount of edges and vertices:

Algorithm 25 Demonstration of ‘create_k2_graph’

```
#include <cassert>

#include "create_k2_graph.h"

void create_k2_graph_demo() noexcept
{
    const auto g = create_k2_graph();
    assert(boost::num_vertices(g) == 2);
    assert(boost::num_edges(g) == 1);
}
```

2.15.4 The .dot file produced

Running a bit ahead, this graph can be converted to the .dot file as shown in algorithm 26:

Algorithm 26 .dot file created from the ‘create_k2_graph’ function (algorithm 24), converted from graph to .dot file using algorithm 48

```
graph G {  
0;  
1;  
0--1 ;  
}
```

From the .dot file one can already see that the graph is undirected, because:

- The first word, ‘graph’, denotes an undirected graph (where ‘digraph’ would have indicated a directional graph)
- The edge between 0 and 1 is written as ‘–’ (where directed connections would be written as ‘->’, ‘<-’ or ‘<>’)

2.15.5 The .svg file produced

Continuing to running a bit ahead, this .dot file can be converted to the .svg as shown in figure 9:

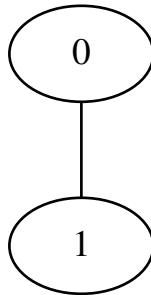


Figure 9: .svg file created from the ‘create_k2_graph’ function (algorithm 24) its .dot file, converted from .dot file to .svg using algorithm 339

Also this figure shows that the graph is undirected, otherwise the edge would have one or two arrow heads. The vertices display the node index, which is the default behavior.

2.16 ► Creating K_3 , a fully connected undirected graph with three vertices

This is an extension of the previous chapter

2.16.1 Graph

To create a fully connected undirected graph with two vertices (also called K_2), one needs two vertices and one (undirected) edge, as depicted in figure 10.

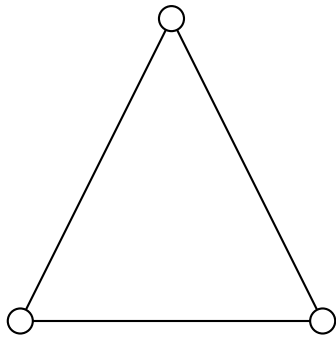


Figure 10: K_3 : a fully connected graph with three edges and vertices

2.16.2 Function to create such a graph

To create K_3 , the following code can be used:

Algorithm 27 Creating K_3 as depicted in figure 10

```
#include <cassert>
#include "create_empty_undirected_graph.h"
#include "create_k3_graph.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS
>
create_k3_graph() noexcept
{
    auto g = create_empty_undirected_graph();
    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);
    const auto vd_c = boost::add_vertex(g);
    const auto aer_a = boost::add_edge(vd_a, vd_b, g);
    assert(aer_a.second);
    const auto aer_b = boost::add_edge(vd_b, vd_c, g);
    assert(aer_b.second);
    const auto aer_c = boost::add_edge(vd_c, vd_a, g);
    assert(aer_c.second);
    return g;
}
```

2.16.3 Creating such a graph

Algorithm 28 demonstrates how to ‘create_k2_graph’ and checks if it has the correct amount of edges and vertices:

Algorithm 28 Demonstration of ‘create_k3_graph’

```
#include "create_k3_graph.h"

void create_k3_graph_demo() noexcept
{
    const auto g = create_k3_graph();
    assert(boost::num_edges(g) == 3);
    assert(boost::num_vertices(g) == 3);
}
```

2.16.4 The .dot file produced

This graph can be converted to the .dot file as shown in algorithm 29:

Algorithm 29 .dot file created from the ‘create_k3_graph’ function (algorithm 27), converted from graph to .dot file using algorithm 48

```
graph G {  
0;  
1;  
2;  
0--1 ;  
1--2 ;  
2--0 ;  
}
```

2.16.5 The .svg file produced

Continuing to running a bit ahead, this .dot file can be converted to the .svg as shown in figure 11:

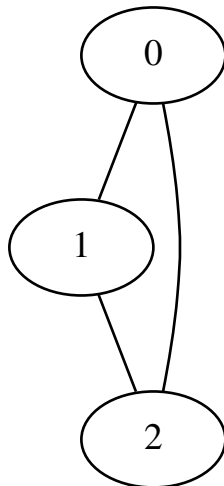


Figure 11: .svg file created from the ‘create_k3_graph’ function (algorithm 27) its .dot file, converted from .dot file to .svg using algorithm 339

2.17 ► Creating a path graph

A path graph is a linear graph without any branches

2.17.1 Graph

Here I show a path graph with four vertices (see figure 12):

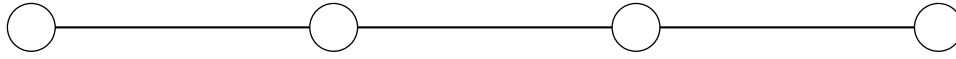


Figure 12: A path graph with four vertices

2.17.2 Function to create such a graph

To create a path graph, the following code can be used:

Algorithm 30 Creating a path graph as depicted in figure 12

```
#include "create_empty_undirected_graph.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS
>
create_path_graph(const int n_vertices) noexcept
{
    assert(n_vertices >= 2);
    auto g = create_empty_undirected_graph();

    auto vd_1 = boost::add_vertex(g);
    for (int i=1; i!=n_vertices; ++i)
    {
        auto vd_2 = boost::add_vertex(g);
        const auto aer = boost::add_edge(vd_1, vd_2, g);
        assert(aer.second);
        vd_1 = vd_2;
    }
    return g;
}
```

2.17.3 Creating such a graph

Algorithm 31 demonstrates how to ‘create_k2_graph’ and checks if it has the correct amount of edges and vertices:

Algorithm 31 Demonstration of ‘create_path_graph’

```
#include <cassert>
#include "create_path_graph.h"

void create_path_graph_demo() noexcept
{
    const auto g = create_path_graph(4);
    assert(boost::num_edges(g) == 3);
    assert(boost::num_vertices(g) == 4);
}
```

2.17.4 The .dot file produced

This graph can be converted to the .dot file as shown in algorithm 32:

Algorithm 32 .dot file created from the ‘create_path_graph’ function (algorithm 30), converted from graph to .dot file using algorithm 48

```
graph G {
0;
1;
2;
3;
0--1 ;
1--2 ;
2--3 ;
}
```

2.17.5 The .svg file produced

The .dot file can be converted to the .svg as shown in figure 13:

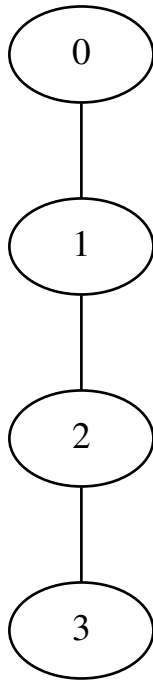


Figure 13: .svg file created from the ‘create_path_graph’ function (algorithm 30) its .dot file, converted from .dot file to .svg using algorithm 339

2.18 ► Creating a Petersen graph

A Petersen graph is the first graph with interesting properties.

2.18.1 Graph

To create a Petersen graph, one needs five vertices and five undirected edges, as depicted in figure 14.

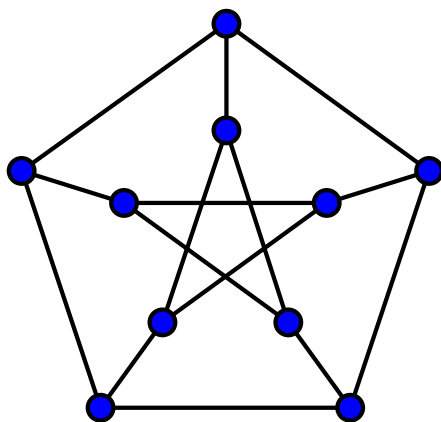


Figure 14: A Petersen graph (from https://en.wikipedia.org/wiki/Petersen_graph)

2.18.2 Function to create such a graph

To create a Petersen graph, the following code can be used:

Algorithm 33 Creating Petersen graph as depicted in figure 14

```
#include <cassert>
#include <vector>
#include "create_empty_undirected_graph.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS
>
create_petersen_graph() noexcept
{
    /*
    using vd = boost::graph_traits<
        boost::adjacency_list<
            boost::vecS,
            boost::vecS,
            boost::undirectedS
        >::vertex_descriptor;
    */
    using vd = decltype(create_empty_undirected_graph())::
        vertex_descriptor;

    auto g = create_empty_undirected_graph();

    std::vector<vd> v; //Outer
    for (int i=0; i!=5; ++i) {
        v.push_back(boost::add_vertex(g));
    }
    std::vector<vd> w; //Inner
    for (int i=0; i!=5; ++i) {
        w.push_back(boost::add_vertex(g));
    }
    //Outer ring
    for (int i=0; i!=5; ++i) {
        const auto aer
            = boost::add_edge(v[i], v[(i + 1) % 5], g);
        assert(aer.second);
    }
    //Spoke
    for (int i=0; i!=5; ++i) {
        const auto aer
            = boost::add_edge(v[i], w[i], g);
        assert(aer.second);
    }
    //Inner pentagram
    for (int i=0; i!=5; ++i) {45
        const auto aer
            = boost::add_edge(w[i], w[(i + 2) % 5], g);
        assert(aer.second);
    }
    return g;
}
```

2.18.3 Creating such a graph

Algorithm 34 demonstrates how to use ‘create_petersen_graph’ and checks if it has the correct amount of edges and vertices:

Algorithm 34 Demonstration of ‘create_k3_graph’

```
#include <cassert>
#include "create_petersen_graph.h"

void create_petersen_graph_demo() noexcept
{
    const auto g = create_petersen_graph();
    assert(boost::num_edges(g) == 15);
    assert(boost::num_vertices(g) == 10);
}
```

2.18.4 The .dot file produced

This graph can be converted to the .dot file as shown in algorithm 35:

Algorithm 35 .dot file created from the ‘create_petersen_graph’ function (algorithm 33), converted from graph to .dot file using algorithm 48

```
graph G {
0;
1;
2;
3;
4;
5;
6;
7;
8;
9;
0--1 ;
1--2 ;
2--3 ;
3--4 ;
4--0 ;
0--5 ;
1--6 ;
2--7 ;
3--8 ;
4--9 ;
5--7 ;
6--8 ;
7--9 ;
8--5 ;
9--6 ;
}
```

2.18.5 The .svg file produced

This .dot file can be converted to the .svg as shown in figure 15:

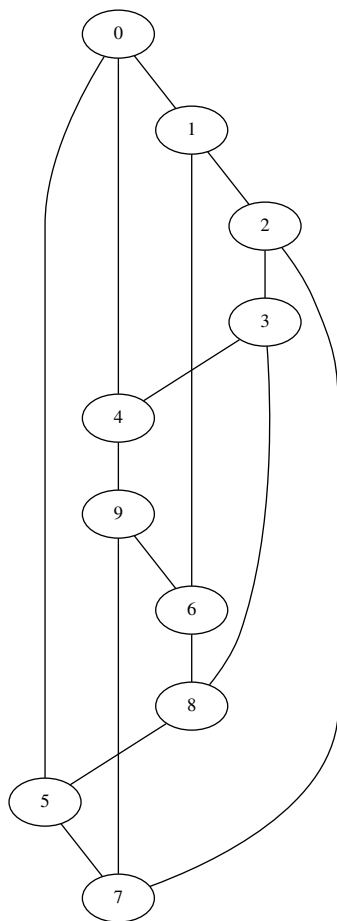


Figure 15: .svg file created from the ‘create_petersen_graph’ function (algorithm 33) its .dot file, converted from .dot file to .svg using algorithm 339

3 Working on graphs without properties

Now that we can build a graph, there are some things we can do.

- Getting the vertices’ out degrees: see chapter 3.1
- Create a direct-neighbour subgraph from a vertex descriptor
- Create all direct-neighbour subgraphs from a graphs
- Saving a graph without properties to .dot file: see chapter 3.7
- Loading an undirected graph without properties from .dot file: see chapter 3.9

- Loading a directed graph without properties from .dot file: see chapter 3.8

3.1 Getting the vertices' out degree

Let's measure the out degree of all vertices in a graph!

The out degree of a vertex is the number of edges that originate at it.

The number of connections is called the 'degree' of the vertex. There are three types of degrees:

- in degree: the number of incoming connections, using 'in_degree' (not 'boost::in_edgree')
- out degree: the number of outgoing connections, using 'out_degree' (not 'boost::out_edgree')
- degree: sum of the in degree and out degree, using 'degree' (not 'boost::edgree')

Algorithm 36 shows how to obtain these:

Algorithm 36 Get the vertices' out degrees

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

template <typename graph>
std::vector<int> get_vertex_out_degrees(
    const graph& g
) noexcept
{
    using vd = typename graph::vertex_descriptor;

    std::vector<int> v(boost::num_vertices(g));
    const auto vip = vertices(g);
    std::transform(vip.first, vip.second, std::begin(v),
        [g](const vd& d) {
            return out_degree(d,g);
        }
    );
    return v;
}
```

The structure of this algorithm is similar to 'get_vertex_descriptors' (algorithm 13), except that the out degrees from the vertex descriptors are stored. The out degree of a vertex iterator is obtained from the function 'out_degree' (not boost::out_degree!).

Albeit that the K_2 graph and the two-state Markov chain are rather simple, we can use it to demonstrate ‘get_vertex_out_degrees’ on, as shown in algorithm 37.

Algorithm 37 Demonstration of the ‘get_vertex_out_degrees’ function

```
#include <cassert>

#include "create_k2_graph.h"
#include "create_markov_chain.h"
#include "get_vertex_out_degrees.h"

void get_vertex_out_degrees_demo() noexcept
{
    const auto g = create_k2_graph();
    const std::vector<int> expected_out_degrees_g{1,1};
    const std::vector<int> vertex_out_degrees_g{
        get_vertex_out_degrees(g)
    };
    assert(expected_out_degrees_g
        == vertex_out_degrees_g
    );

    const auto h = create_markov_chain();
    const std::vector<int> expected_out_degrees_h{2,2};
    const std::vector<int> vertex_out_degrees_h{
        get_vertex_out_degrees(h)
    };
    assert(expected_out_degrees_h
        == vertex_out_degrees_h
    );
}
```

It is expected that K_2 has one out-degree for every vertex, where the two-state Markov chain is expected to have two out-degrees per vertex.

3.2 ► Is there an edge between two vertices?

If you have two vertex descriptors, you can check if these are connected by an edge:

Algorithm 38 Check if there exists an edge between two vertices

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

template <typename graph>
bool has_edge_between_vertices(
    const typename boost::graph_traits<graph>::
        vertex_descriptor& vd_1,
    const typename boost::graph_traits<graph>::
        vertex_descriptor& vd_2,
    const graph& g
) noexcept
{
    return edge(vd_1, vd_2, g).second;
}
```

This code uses the function ‘edge’ (not `boost::edge`: it returns a pair consisting of an edge descriptor and a boolean indicating if it is a valid edge descriptor. The boolean will be true if there exists an edge between the two vertices and false if not.

The demo shows that there is an edge between the two vertices of a K_2 graph, but there are no self-loops (edges that original and end at the same vertex).

Algorithm 39 Demonstration of the ‘has_edge_between_vertices’ function

```
#include <cassert>
#include "create_k2_graph.h"
#include "has_edge_between_vertices.h"

void has_edge_between_vertices_demo() noexcept
{
    const auto g = create_k2_graph();
    const auto vd_1 = *vertices(g).first;
    const auto vd_2 = *(++vertices(g).first);
    assert( has_edge_between_vertices(vd_1, vd_2, g));
    assert(! has_edge_between_vertices(vd_1, vd_1, g));
}
```

3.3 ► Get the edge between two vertices

If you have two vertex descriptors, you can use these to find the edge between them.

Algorithm 40 Get the edge between two vertices

```
#include <boost/graph/adjacency_list.hpp>
#include "has_edge_between_vertices.h"

template <
    typename graph,
    typename vertex_descriptor
>
typename boost::graph_traits<graph>::edge_descriptor
get_edge_between_vertices(
    const vertex_descriptor& vd_from,
    const vertex_descriptor& vd_to,
    const graph& g
) noexcept
{
    assert(has_edge_between_vertices(vd_from, vd_to, g));
    const auto er = edge(vd_from, vd_to, g);
    assert(er.second);
    return er.first;
}
```

This code does assume that there is an edge between the two vertices.

The demo shows how to get the edge between two vertices, deleting it, and checking for success.

Algorithm 41 Demonstration of the ‘get_edge_between_vertices’ function

```
#include <cassert>
#include "create_k2_graph.h"
#include "get_edge_between_vertices.h"

void get_edge_between_vertices_demo() noexcept
{
    auto g = create_k2_graph();
    const auto vd_1 = *vertices(g).first;
    const auto vd_2 = *(++vertices(g).first);
    assert(has_edge_between_vertices(vd_1, vd_2, g));
    const auto ed = get_edge_between_vertices(vd_1, vd_2, g);
    boost::remove_edge(ed, g);
    assert(boost::num_edges(g) == 0);
}
```

3.4 ► Create a direct-neighbour subgraph from a vertex descriptor

Suppose you have a vertex of interest its vertex descriptor. Let's say you want to get a subgraph of that vertex and its direct neighbours only. This means that all vertices of that subgraph are adjacent vertices and that the edges go either from focal vertex to its neighbours, or from adjacent vertex to adjacent neighbour.

Here is the 'create_direct_neighbour_subgraph' code:

Algorithm 42 Get the direct-neighbour subgraph from a vertex descriptor

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

template <typename graph, typename vertex_descriptor>
graph create_direct_neighbour_subgraph(
    const vertex_descriptor& vd,
    const graph& g
)
{
    graph h;

    std::map<vertex_descriptor, vertex_descriptor> m;
    {
        const auto vd_h = boost::add_vertex(h);
        m.insert(std::make_pair(vd, vd_h));
    }
    //Copy vertices
    {
        const auto vdsi = boost::adjacent_vertices(vd, g);
        std::transform(vdsi.first, vdsi.second,
            std::inserter(m, std::begin(m)),
            [&h](const vertex_descriptor& d)
            {
                const auto vd_h = boost::add_vertex(h);
                return std::make_pair(d, vd_h);
            }
        );
    }
    //Copy edges
    {
        const auto eip = edges(g);
        const auto j = eip.second;
        for (auto i = eip.first; i!=j; ++i)
        {
            const auto vd_from = source(*i, g);
            const auto vd_to = target(*i, g);
            if (m.find(vd_from) == std::end(m)) continue;
            if (m.find(vd_to) == std::end(m)) continue;
            const auto aer = boost::add_edge(m[vd_from], m[vd_to], h);
            assert(aer.second);
        }
    }
    return h;
}
```

This demonstration code shows that the direct-neighbour graph of each vertex of a K_2 graphs is ... a K_2 graph!

Algorithm 43 Demo of the ‘create_direct_neighbour_subgraph’ function

```
#include "create_direct_neighbour_subgraph.h"
#include "create_k2_graph.h"

void create_direct_neighbour_subgraph_demo() noexcept
{
    const auto g = create_k2_graph();
    const auto vip = vertices(g);
    const auto j = vip.second;
    for (auto i=vip.first; i!=j; ++i) {
        const auto h = create_direct_neighbour_subgraph(
            *i, g
        );
        assert(boost::num_vertices(h) == 2);
        assert(boost::num_edges(h) == 1);
    }
}
```

3.5 ► Creating all direct-neighbour subgraphs from a graph without properties

Using the previous function, it is easy to create all direct-neighbour subgraphs from a graph without properties:

Algorithm 44 Create all direct-neighbour subgraphs from a graph without properties

```
#include <vector>
#include "create_direct_neighbour_subgraph.h"

template <typename graph>
std::vector<graph> create_all_direct_neighbour_subgraphs(
    const graph g
) noexcept
{
    using vd = typename graph::vertex_descriptor;

    std::vector<graph> v;
    v.resize(boost::num_vertices(g));
    const auto vip = vertices(g);
    std::transform(
        vip.first, vip.second,
        std::begin(v),
        [g](const vd& d)
        {
            return create_direct_neighbour_subgraph(
                d, g
            );
        }
    );
    return v;
}
```

This demonstration code shows that all two direct-neighbour graphs of a K_2 graphs are ... K_2 graphs!

Algorithm 45 Demo of the ‘create_all_direct_neighbour_subgraphs’ function

```
#include <cassert>
#include "create_all_direct_neighbour_subgraphs.h"
#include "create_k2_graph.h"

void create_all_direct_neighbour_subgraphs_demo()
    noexcept
{
    const auto v
        = create_all_direct_neighbour_subgraphs(
            create_k2_graph());
    assert(v.size() == 2);
    for (const auto g: v)
    {
        assert(boost::num_vertices(g) == 2);
        assert(boost::num_edges(g) == 1);
    }
}
```

3.6 ► Are two graphs isomorphic?

You may want to check if two graphs are isomorphic. That is: if they have the same shape.

Algorithm 46 Check if two graphs are isomorphic

```
#include <boost/graph/isomorphism.hpp>

template <typename graph1, typename graph2>
bool is_isomorphic(
    const graph1 g,
    const graph2 h
) noexcept
{
    return boost::isomorphism(g, h);
}
```

This demonstration code shows that a K_3 graph is not equivalent to a 3-vertices path graph:

Algorithm 47 Demo of the ‘is_isomorphic’ function

```
#include <cassert>
#include "create_path_graph.h"
#include "create_k3_graph.h"
#include "is_isomorphic.h"

void is_isomorphic_demo() noexcept
{
    const auto g = create_path_graph(3);
    const auto h = create_k3_graph();
    assert( is_isomorphic(g,g));
    assert(!is_isomorphic(g,h));
}
```

3.7 Saving a graph to a .dot file

Graph are easily saved to a file, thanks to Graphviz. Graphviz (short for Graph Visualization Software) is a package of open-source tools for drawing graphs. It uses the DOT language for describing graphs, and these are commonly stored in (plain-text) .dot files (I show .dot file of every non-empty graph created, e.g. chapters 2.14.4 and 2.15.4)

Algorithm 48 Saving a graph to a .dot file

```
#include <fstream>
#include <boost/graph/graphviz.hpp>

template <typename graph>
void save_graph_to_dot(
    const graph& g,
    const std::string& filename
) noexcept
{
    std::ofstream f(filename);
    boost::write_graphviz(f,g);
}
```

All the code does is create an `std::ofstream` (an output-to-file stream) and use `boost::write_graphviz` to write the DOT description of our graph to that stream. Instead of ‘`std::ofstream`’, one could use `std::cout` (a related output stream) to display the DOT language on screen directly.

Algorithm 49 shows how to use the ‘`save_graph_to_dot`’ function:

Algorithm 49 Demonstration of the ‘save_graph_to_dot’ function

```
#include "create_k2_graph.h"
#include "create_markov_chain.h"
#include "save_graph_to_dot.h"

void save_graph_to_dot_demo() noexcept
{
    const auto g = create_k2_graph();
    save_graph_to_dot(g, "create_k2_graph.dot");

    const auto h = create_markov_chain();
    save_graph_to_dot(h, "create_markov_chain.dot");
}
```

When using the ‘save_graph_to_dot’ function (algorithm 48), only the structure of the graph is saved: all other properties like names are not stored. Algorithm 105 shows how to do so.

3.8 Loading a directed graph from a .dot

When loading a graph from file, one needs to specify a type of graph. In this example, an directed graph is loaded, as shown in algorithm 50:

Algorithm 50 Loading a directed graph from a .dot file

```
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "create_empty_directed_graph.h"
#include "is_regular_file.h"

boost::adjacency_list<>
load_directed_graph_from_dot(
    const std::string& dot_filename
)
{
    assert(is_regular_file(dot_filename));
    std::ifstream f(dot_filename.c_str());
    auto g = create_empty_directed_graph();
    boost::dynamic_properties dp(
        boost::ignore_other_properties
    );
    boost::read_graphviz(f, g, dp);
    return g;
}
```

In this algorithm, first it is checked if the file to load exists, using the ‘is_regular_file’ function (algorithm 340), after which an `std::ifstream` is opened. Then an empty directed graph is created, which saves us writing down the template arguments explicitly. Then, a `boost::dynamic_properties` is created with the ‘boost::ignore_other_properties’ in its constructor (using a default constructor here results in the run-time error ‘property not found: node_id’, see chapter 24.5). From this and the empty graph, ‘boost::read_graphviz’ is called to build up the graph.

Algorithm 51 shows how to use the ‘load_directed_graph_from_dot’ function:

Algorithm 51 Demonstration of the ‘load_directed_graph_from_dot’ function

```
#include <cassert>
#include "create_markov_chain.h"
#include "load_directed_graph_from_dot.h"
#include "save_graph_to_dot.h"

void load_directed_graph_from_dot_demo() noexcept
{
    using boost::num_edges;
    using boost::num_vertices;

    const auto g = create_markov_chain();
    const std::string filename{
        "create_markov_chain.dot"
    };
    save_graph_to_dot(g, filename);
    const auto h = load_directed_graph_from_dot(filename);
    assert(num_edges(g) == num_edges(h));
    assert(num_vertices(g) == num_vertices(h));
}
```

This demonstration shows how the Markov chain is created using the ‘create_markov_chain_graph’ function (algorithm 21), saved and then loaded. The loaded graph is then checked to be a two-state Markov chain.

3.9 Loading an undirected graph from a .dot file

Loading an undirected graph from a .dot file is very similar to loading a directed graph from a .dot file, as shown in chapter 3.8. Algorithm 52 show how to do so:

Algorithm 52 Loading an undirected graph from a .dot file

```
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "create_empty_undirected_graph.h"
#include "is_regular_file.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS
>
load_undirected_graph_from_dot(
    const std::string& dot_filename
)
{
    assert(is_regular_file(dot_filename));
    std::ifstream f(dot_filename.c_str());
    auto g = create_empty_undirected_graph();
    boost::dynamic_properties p(
        boost::ignore_other_properties
    );
    boost::read_graphviz(f, g, p);
    return g;
}
```

The only difference with loading a directed graph, is that the initial empty graph is undirected instead. Chapter 3.8 describes the rationale of this function.

Algorithm 53 shows how to use the ‘load_undirected_graph_from_dot’ function:

Algorithm 53 Demonstration of the ‘load_undirected_graph_from_dot’ function

```
#include <cassert>
#include "create_k2_graph.h"
#include "load_undirected_graph_from_dot.h"
#include "save_graph_to_dot.h"

void load_undirected_graph_from_dot_demo() noexcept
{
    using boost::num_edges;
    using boost::num_vertices;

    const auto g = create_k2_graph();
    const std::string filename{"create_k2_graph.dot"};
    save_graph_to_dot(g, filename);
    const auto h
        = load_undirected_graph_from_dot(filename);
    assert(num_edges(g) == num_edges(h));
    assert(num_vertices(g) == num_vertices(h));
}
```

This demonstration shows how the K_2 graph is created using the ‘create_k2_graph’ function (algorithm 24), saved and then loaded. The loaded graph is checked to be a K_2 graph.

4 Building graphs with named vertices

Up until now, the graphs created have had edges and vertices without any property. In this chapter, graphs will be created, in which the vertices can have a name. This name will be of the `std::string` data type, but other types are possible as well. There are many more built-in properties edges and nodes can have (see chapter 25.1 for a list).

In this chapter, we will build the following graphs:

- An empty directed graph that allows for vertices with names: see chapter 4.1
- An empty undirected graph that allows for vertices with names: see chapter 4.2
- Two-state Markov chain with named vertices: see chapter 4.5
- K_2 with named vertices: see chapter 4.6

In the process, some basic (sometimes bordering trivial) functions are shown:

- Adding a named vertex: see chapter 4.3
- Getting the vertices' names: see chapter 4.4

After this chapter you may want to:

- Building graphs with named edges and vertices: see chapter 6
- Building graphs with bundled vertices: see chapter 8
- Building graphs with custom vertices: see chapter 12
- Building graphs with a graph name: see chapter 20

4.1 Creating an empty directed graph with named vertices

Let's create a trivial empty directed graph, in which the vertices can have a name:

Algorithm 54 Creating an empty directed graph with named vertices

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

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    boost::property<
        boost::vertex_name_t, std::string
    >
>
>
create_empty_directed_named_vertices_graph() noexcept
{
    return {};
}
```

Instead of using a `boost::adjacency_list` with default template argument, we will now have to specify four template arguments, where we only set the fourth to a non-default value.

Note there is some flexibility in this function: the data type of the vertex names is set to `std::string` by default, but can be of any other type if desired.

This graph:

- has its out edges stored in a `std::vector` (due to the first `boost::vecS`)
- has its vertices stored in a `std::vector` (due to the second `boost::vecS`)

- is directed (due to the `boost::directedS`)
- The vertices have one property: they have a name, which is of data type `std::string` (due to the `boost::property<boost::vertex_name_t, std::string>`)
- Edges and graph have no properties
- Edges are stored in a `std::list`

The `boost::adjacency_list` has a new, fourth template argument '`boost::property<boost::vertex_name_t, std::string>`'. This can be read as: "vertices have the property '`boost::vertex_name_t`', that is of data type '`std::string`'. Or simply: "vertices have a name that is stored as a `std::string`".

Algorithm 55 shows how to create such a graph:

| | | | | | |
|------------------|-----------|---------------|----|-----|---|
| Algorithm | 55 | Demonstration | of | the | 'create_empty_directed_named_vertices_graph' function |
|------------------|-----------|---------------|----|-----|---|

```
#include <cassert>
#include <boost/graph/adjacency_list.hpp>
#include "create_empty_directed_named_vertices_graph.h"

void create_empty_named_directed_vertices_graph_demo()
    noexcept
{
    const auto g
        = create_empty_directed_named_vertices_graph();
    assert(boost::num_vertices(g) == 0);
    assert(boost::num_edges(g) == 0);
}
```

4.2 Creating an empty undirected graph with named vertices

Let's create a trivial empty undirected graph, in which the vertices can have a name:

Algorithm 56 Creating an empty undirected graph with named vertices

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

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
        boost::vertex_name_t, std::string
    >
>
>
create_empty_undirected_named_vertices_graph() noexcept
{
    return {};
}
```

This code is very similar to the code described in chapter 4.1, except that the directedness (the third template argument) is undirected (due to the `boost::undirectedS`). See chapter 4.1 for most of the explanation.

Algorithm 57 shows how to create such a graph:

Algorithm 57 Demonstration of the ‘create_empty_undirected_named_vertices_graph’ function

```
#include <cassert>
#include <boost/graph/adjacency_list.hpp>
#include "create_empty_undirected_named_vertices_graph.h"

void create_empty_undirected_named_vertices_graph_demo()
    noexcept
{
    const auto g
        = create_empty_undirected_named_vertices_graph();
    assert(boost::num_vertices(g) == 0);
    assert(boost::num_edges(g) == 0);
}
```

4.3 Add a vertex with a name

Adding a vertex without a name was trivially easy (see chapter 2.5). Adding a vertex with a name takes slightly more work, as shown by algorithm 58:

Algorithm 58 Adding a vertex with a name

```
#include <string>
#include <type_traits>
#include <boost/graph/adjacency_list.hpp>

template <typename graph>
typename boost::graph_traits<graph>::vertex_descriptor
add_named_vertex(
    const std::string& vertex_name,
    graph& g
) noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const"
    );

    const auto vd = boost::add_vertex(g);
    auto vertex_name_map = get(
        boost::vertex_name, g
    );
    put(vertex_name_map, vd, vertex_name);
    return vd;
}
```

Instead of calling ‘boost::add_vertex’ with an additional argument containing the name of the vertex⁶, multiple things need to be done:

First, the `static_assert` at the top of the function checks during compiling if the function is called with a non-const graph. One can freely omit this `static_assert`: you will get a compiler error anyways, be it a less helpful one.

When adding a new vertex to the graph, the vertex descriptor (as described in chapter 2.6) is stored.

The name map is obtained from the graph using ‘get’. ‘get’ (not `boost::get`) allow to obtain a property map. In this case, ‘get(boost::vertex_name,g)’ denotes that we want to obtain the property map associated with ‘boost::vertex_name’ from the graph. ‘get’ has no ‘boost::’ prepending it, as it lives in the same (global) namespace the function is in. Using ‘boost::get’ will not compile.

With a name map and a vertex descriptor, the name of a vertex can be set using ‘put’ (not `boost::put`). ‘put’ is the opposite of ‘get’. In this case ‘put(vertex_name_map, vd, vertex_name)’ is read as: in the vertex name map, look up the spot where the vertex we have the descriptor of, and put the new vertex name there. An alternative syntax is ‘vertex_name_map[vd] =

⁶I am unsure if this would have been a good interface. I am sure I expected this interface myself. I do see a problem with multiple properties and the order of initialization, but initialization could simply follow the same order as the the property list.

vertex_name’. Because ‘put’ is more general, it is chosen to be the preferred syntax for this tutorial.

Using ‘add_named_vertex’ is straightforward, as demonstrated by algorithm 59.

Algorithm 59 Demonstration of ‘add_named_vertex’

```
#include <cassert>
#include "add_named_vertex.h"
#include "create_empty_undirected_named_vertices_graph.h"

void add_named_vertex_demo() noexcept
{
    auto g
        = create_empty_undirected_named_vertices_graph();
    add_named_vertex("Lex", g);
    assert(boost::num_vertices(g) == 1);
}
```

4.4 Getting the vertices’ names

When the vertices of a graph have named vertices, one can extract them as such:

Algorithm 60 Get the vertices' names

```
#include <string>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/properties.hpp>
#include <boost/graph/graph_traits.hpp>

template <typename graph>
std::vector<std::string> get_vertex_names(
    const graph& g
) noexcept
{
    using vd = typename graph::vertex_descriptor;

    std::vector<std::string> v(boost::num_vertices(g));
    const auto vip = vertices(g);
    std::transform(vip.first, vip.second, std::begin(v),
        [g](const vd& d)
        {
            const auto vertex_name_map = get(
                boost::vertex_name, g
            );
            return get(vertex_name_map, d);
        }
    );
    return v;
}
```

This code is very similar to ‘get_vertex_out_degrees’ (algorithm 36), as also there we iterated through all vertices, accessing all vertex descriptors sequentially.

The names of the vertices are obtained from a `boost::property_map` and then put into a `std::vector`.

The order of the vertex names may be different after saving and loading.

When trying to get the vertices' names from a graph without vertices with names, you will get the error ‘formed reference to void’ (see chapter 24.1).

Algorithm 61 shows how to add two named vertices, and check if the added names are retrieved as expected.

Algorithm 61 Demonstration of ‘get_vertex_names’

```
#include <cassert>

#include "add_named_vertex.h"
#include "create_empty_undirected_named_vertices_graph.h"
#include "get_vertex_names.h"

void get_vertex_names_demo() noexcept
{
    auto g
        = create_empty_undirected_named_vertices_graph();
    const std::string vertex_name_1{"Chip"};
    const std::string vertex_name_2{"Chap"};
    add_named_vertex(vertex_name_1, g);
    add_named_vertex(vertex_name_2, g);
    const std::vector<std::string> expected_names{
        vertex_name_1, vertex_name_2
    };
    const std::vector<std::string> vertex_names{
        get_vertex_names(g)
    };
    assert(expected_names == vertex_names);
}
```

4.5 Creating a Markov chain with named vertices

Let’s create a directed non-empty graph with named vertices!

4.5.1 Graph

We extend the Markov chain of chapter 2.14 by naming the vertices ‘Good’ and ‘Not bad’, as depicted in figure 16:

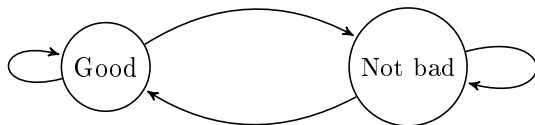


Figure 16: A two-state Markov chain where the vertices have texts

The vertex names are nonsensical, but I choose these for a reason: one name is only one word, the other has two words (as it contains a space). This will have implications for file I/O.

4.5.2 Function to create such a graph

To create this Markov chain, the following code can be used:

Algorithm 62 Creating a Markov chain with named vertices as depicted in figure 16

```
#include <cassert>
#include "add_named_vertex.h"
#include "create_empty_directed_named_vertices_graph.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    boost::property<boost::vertex_name_t, std::string>
>
create_named_vertices_markov_chain() noexcept
{
    auto g
        = create_empty_directed_named_vertices_graph();
    const auto vd_a = add_named_vertex("Good", g);
    const auto vd_b = add_named_vertex("Not_bad", g);
    const auto aer_aa = boost::add_edge(vd_a, vd_a, g);
    assert(aer_aa.second);
    const auto aer_ab = boost::add_edge(vd_a, vd_b, g);
    assert(aer_ab.second);
    const auto aer_ba = boost::add_edge(vd_b, vd_a, g);
    assert(aer_ba.second);
    const auto aer_bb = boost::add_edge(vd_b, vd_b, g);
    assert(aer_bb.second);
    return g;
}
```

Most of the code is a repeat of algorithm 21, ‘create_markov_chain_graph’. In the end of the function body, the names are obtained as a `boost::property_map` and set to the desired values.

4.5.3 Creating such a graph

Also the demonstration code (algorithm 63) is very similar to the demonstration code of the ‘create_markov_chain_graph’ function (algorithm 22).

Algorithm 63 Demonstrating the ‘create_named_vertices_markov_chain’ function

```
#include <cassert>

#include "create_named_vertices_markov_chain.h"
#include "get_vertex_names.h"

void create_named_vertices_markov_chain_demo() noexcept
{
    const auto g
        = create_named_vertices_markov_chain();
    const std::vector<std::string> expected_names{
        "Good", "Not_bad"
    };
    const std::vector<std::string> vertex_names{
        get_vertex_names(g)
    };
    assert(expected_names == vertex_names);
}
```

4.5.4 The .dot file produced

Because the vertices now have a name, this should be visible in the .dot file:

Algorithm 64 .dot file created from the ‘create_named_vertices_markov_chain’ function (algorithm 62), converted from graph to .dot file using algorithm 48

```
digraph G {
0[label=Good];
1[label="Not bad"];
0->0 ;
0->1 ;
1->0 ;
1->1 ;
}
```

As one can see, the names are stored as a label. Note that if a vertex name contains a space, the name will be surrounded by quotes, for example ‘1[label="Not bad"]’.

4.5.5 The .svg file produced

Now that the vertices have names, this should be reflected in the .svg:

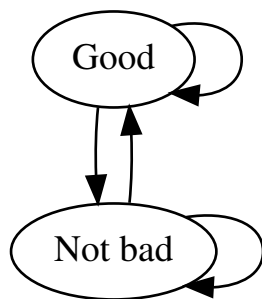


Figure 17: .svg file created from the ‘create_named_vertices_markov_chain’ function (algorithm 62) its .dot file, converted from .dot file to .svg using algorithm 339

The .svg now shows the vertex names, instead of the vertex indices.

4.6 Creating K_2 with named vertices

Let’s create an undirected non-empty graph with named vertices!

4.6.1 Graph

We extend K_2 of chapter 2.15 by naming the vertices ‘Me’ and ‘My computer’, as depicted in figure 18:

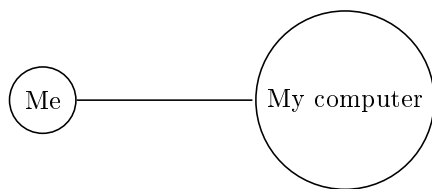


Figure 18: K_2 : a fully connected graph with two named vertices

4.6.2 Function to create such a graph

To create K_2 , the following code can be used:

Algorithm 65 Creating K_2 with named vertices as depicted in figure 18

```
#include <cassert>
#include "create_empty_undirected_named_vertices_graph.h"
#include "add_named_vertex.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<boost::vertex_name_t, std::string>
>
create_named_vertices_k2_graph() noexcept
{
    auto g
        = create_empty_undirected_named_vertices_graph();
    const auto vd_a = add_named_vertex(
        "Me", g
    );
    const auto vd_b = add_named_vertex(
        "My_computer", g
    );
    const auto aer = boost::add_edge(vd_a, vd_b, g);
    assert(aer.second);
    return g;
}
```

Most of the code is a repeat of algorithm 24. In the end, the names are obtained as a `boost::property_map` and set to the desired names.

4.6.3 Creating such a graph

Also the demonstration code (algorithm 66) is very similar to the demonstration code of the ‘`create_k2_graph` function’ (algorithm 24).

Algorithm 66 Demonstrating the ‘create_k2_graph’ function

```
#include <cassert>

#include "create_named_vertices_k2_graph.h"
#include "get_vertex_names.h"

void create_named_vertices_k2_graph_demo() noexcept
{
    const auto g = create_named_vertices_k2_graph();
    const std::vector<std::string> expected_names{"Me", "My
        _computer"};
    const std::vector<std::string> vertex_names =
        get_vertex_names(g);
    assert(expected_names == vertex_names);
}
```

4.6.4 The .dot file produced

Because the vertices now have a name, this should be visible in the .dot file:

Algorithm 67 .dot file created from the ‘create_named_vertices_k2’ function (algorithm 65), converted from graph to .dot file using algorithm 105

```
graph G {
0[label=Me];
1[label="My computer"];
0--1 ;
}
```

As one can see, the names are stored as a label. Note that if a vertex name contains a space, the name will be surrounded by quotes, for example ‘1[label="My computer"]’.

4.6.5 The .svg file produced

Now that the vertices have names, this should be reflected in the .svg:

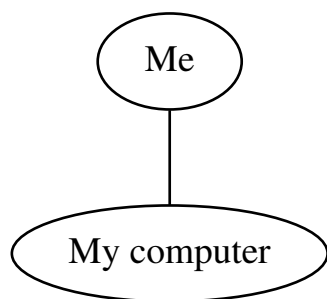


Figure 19: .svg file created from the ‘create_named_vertices_k2_graph’ function (algorithm 62) its .dot file, converted from .dot file to .svg using algorithm 105

The .svg now shows the vertex names, instead of the vertex indices.

4.7 ► Creating K_3 with named vertices

Here we create a K_3 graph with names vertices

4.7.1 Graph

Here I show a K_3 graph with named vertices (see figure 20):

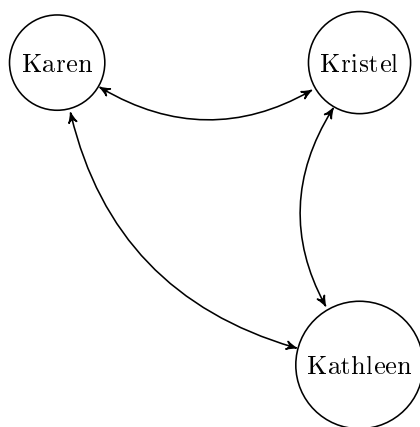


Figure 20: A K_3 graph with named vertices

4.7.2 Function to create such a graph

To create a K_3 graph with named vertices, the following code can be used:

Algorithm 68 Creating a K_3 graph as depicted in figure 20

```
#include <cassert>
#include "create_empty_undirected_named_vertices_graph.h"
#include "add_named_vertex.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<boost::vertex_name_t, std::string>
>
create_named_vertices_k3_graph() noexcept
{
    auto g
        = create_empty_undirected_named_vertices_graph();
    const auto vd_a = add_named_vertex(
        "Karen", g
    );
    const auto vd_b = add_named_vertex(
        "Kristel", g
    );
    const auto vd_c = add_named_vertex(
        "Kathleen", g
    );
    const auto aer_a = boost::add_edge(vd_a, vd_b, g);
    assert(aer_a.second);
    const auto aer_b = boost::add_edge(vd_b, vd_c, g);
    assert(aer_b.second);
    const auto aer_c = boost::add_edge(vd_c, vd_a, g);
    assert(aer_c.second);
    return g;
}
```

4.7.3 Creating such a graph

Algorithm 69 demonstrates how to create a K_3 graph with named vertices and checks if it has the correct amount of edges and vertices:

Algorithm 69 Demonstration of ‘create_named_vertices_k3_graph’

```
#include <cassert>

#include "create_named_vertices_k3_graph.h"
#include "get_vertex_names.h"

void create_named_vertices_k3_graph_demo() noexcept
{
    const auto g = create_named_vertices_k3_graph();
    const std::vector<std::string> expected_names{
        "Karen", "Kristel", "Kathleen"
    };
    const std::vector<std::string> vertex_names =
        get_vertex_names(g);
    assert(expected_names == vertex_names);
}
```

4.7.4 The .dot file produced

This graph can be converted to the .dot file as shown in algorithm 70:

Algorithm 70 .dot file created from the ‘create_named_vertices_k3_graph’ function (algorithm 68), converted from graph to .dot file using algorithm 48

```
graph G {
0[label=Karen];
1[label=Kristel];
2[label=Kathleen];
0--1 ;
1--2 ;
2--0 ;
}
```

4.7.5 The .svg file produced

The .dot file can be converted to the .svg as shown in figure 21:

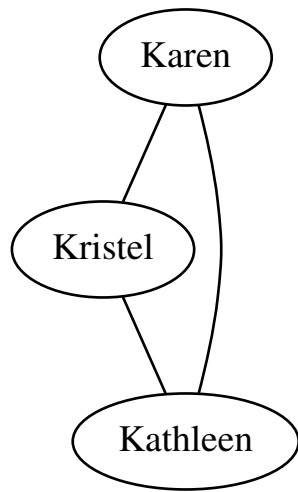


Figure 21: .svg file created from the ‘create_named_vertices_k3_graph’ function (algorithm 68) its .dot file, converted from .dot file to .svg using algorithm 339

4.8 ► Creating a path graph with named vertices

Here we create a path graph with names vertices

4.8.1 Graph

Here I show a path graph with four vertices (see figure 22):

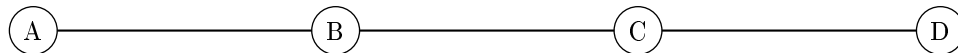


Figure 22: A path graph with four vertices

4.8.2 Function to create such a graph

To create a path graph, the following code can be used:

Algorithm 71 Creating a path graph as depicted in figure 22

```
#include <vector>
#include "add_named_vertex.h"
#include "create_empty_undirected_named_vertices_graph.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
        boost::vertex_name_t, std::string
    >
>
>
create_named_vertices_path_graph(
    const std::vector<std::string>& names
) noexcept
{
    auto g = create_empty_undirected_named_vertices_graph()
        ;
    if (names.size() == 0) { return g; }
    auto vd_1 = add_named_vertex(*names.begin(), g);
    if (names.size() == 1) return g;
    const auto j = std::end(names);
    auto i = std::begin(names);
    for (++i; i!=j; ++i) //Skip first
    {
        auto vd_2 = add_named_vertex(*i, g);
        const auto aer = boost::add_edge(vd_1, vd_2, g);
        assert(aer.second);
        vd_1 = vd_2;
    }
    return g;
}
```

4.8.3 Creating such a graph

Algorithm 72 demonstrates how to create a path graph with named vertices and checks if it has the correct amount of edges and vertices:

Algorithm 72 Demonstration of ‘create_named_vertices_path_graph’

```
#include <cassert>
#include "create_named_vertices_path_graph.h"

void create_named_vertices_path_graph_demo() noexcept
{
    const auto g = create_named_vertices_path_graph(
        {"A", "B", "C", "D"}
    );
    assert(boost::num_edges(g) == 3);
    assert(boost::num_vertices(g) == 4);
}
```

4.8.4 The .dot file produced

This graph can be converted to the .dot file as shown in algorithm 73:

Algorithm 73 .dot file created from the ‘create_named_vertices_path_graph’
function (algorithm 71), converted from graph to .dot file using algorithm 48

4.8.5 The .svg file produced

The .dot file can be converted to the .svg as shown in figure 23:

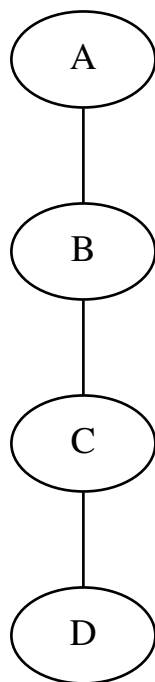


Figure 23: .svg file created from the ‘create_named_vertices_path_graph’ function (algorithm 71) its .dot file, converted from .dot file to .svg using algorithm 339

4.9 ► Creating a Petersen graph with named vertices

Here we create a Petersen graph with names vertices.

4.9.1 Graph

Here I show a Petersen graph (see figure 24):

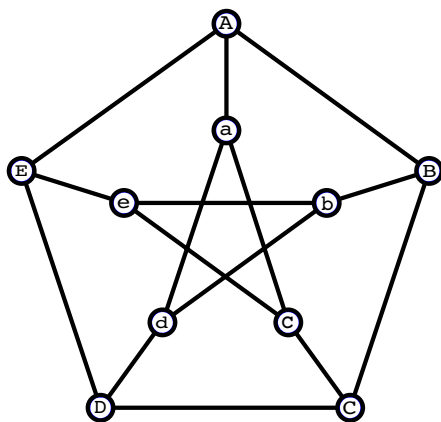


Figure 24: A Petersen graph with named vertices (modified from https://en.wikipedia.org/wiki/Petersen_graph)

4.9.2 Function to create such a graph

To create a Petersen graph with named vertices, the following code can be used:

Algorithm 74 Creating a Petersen graph as depicted in figure 24

```
#include <cassert>
#include <vector>
#include "add_named_vertex.h"
#include "create_empty_undirected_named_vertices_graph.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
        boost::vertex_name_t, std::string
    >
>
>
create_named_vertices_petersen_graph() noexcept
{
    auto g = create_empty_undirected_named_vertices_graph();
    ;
    using vd = decltype(
        create_empty_undirected_named_vertices_graph())::
        vertex_descriptor;

    std::vector<vd> v; //Outer
    for (int i=0; i!=5; ++i) {
        v.push_back(
            add_named_vertex(std::string(1, 'A' + i), g)
        );
    }
    std::vector<vd> w; //Inner
    for (int i=0; i!=5; ++i) {
        w.push_back(
            add_named_vertex(std::string(1, 'a' + i), g)
        );
    }
    //Outer ring
    for (int i=0; i!=5; ++i) {
        const auto aer
            = boost::add_edge(v[i], v[(i + 1) % 5], g);
        assert(aer.second);
    }
    //Spoke
    for (int i=0; i!=5; ++i) {
        const auto aer
            = boost::add_edge(v[i], w[i], g);
        assert(aer.second);
    }
    //Inner pentagram
    for (int i=0; i!=5; ++i) {
        const auto aer
            = boost::add_edge(w[i], w[(i + 2) % 5], g);
        assert(aer.second);
    }
    return g;
}
```

4.9.3 Creating such a graph

Algorithm 75 demonstrates how to create a path graph with named vertices and checks if it has the correct amount of edges and vertices:

Algorithm 75 Demonstration of ‘create_named_vertices_petersen_graph’

```
#include <cassert>
#include "create_named_vertices_petersen_graph.h"

void create_named_vertices_petersen_graph_demo() noexcept
{
    const auto g = create_named_vertices_petersen_graph();
    assert(boost::num_edges(g) == 15);
    assert(boost::num_vertices(g) == 10);
}
```

4.9.4 The .dot file produced

This graph can be converted to the .dot file as shown in algorithm 76:

Algorithm 76 .dot file created from the ‘create_named_vertices_petersen_graph’ function (algorithm 74), converted from graph to .dot file using algorithm 48

```
graph G {
0[label=A];
1[label=B];
2[label=C];
3[label=D];
4[label=E];
5[label=a];
6[label=b];
7[label=c];
8[label=d];
9[label=e];
0--1 ;
1--2 ;
2--3 ;
3--4 ;
4--0 ;
0--5 ;
1--6 ;
2--7 ;
3--8 ;
4--9 ;
5--7 ;
6--8 ;
7--9 ;
8--5 ;
9--6 ;
}
```

4.9.5 The .svg file produced

The .dot file can be converted to the .svg as shown in figure 25:

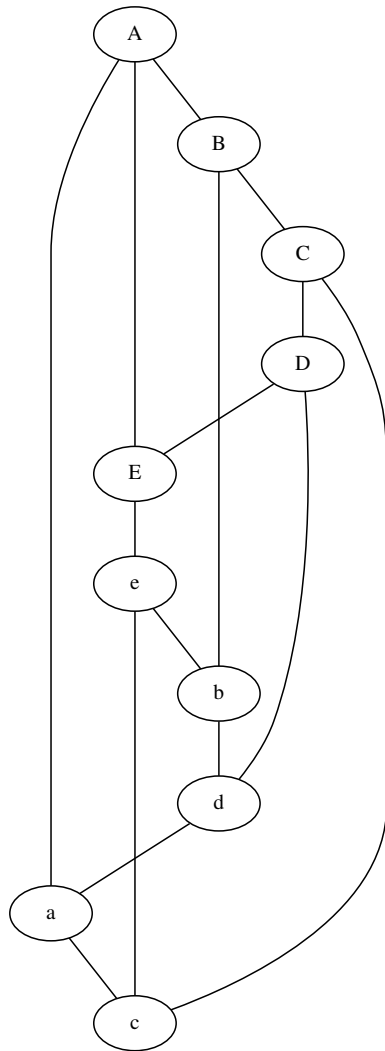


Figure 25: .svg file created from the ‘create_named_vertices_petersen_graph’ function (algorithm 74) its .dot file, converted from .dot file to .svg using algorithm 339

5 Working on graphs with named vertices

When vertices have names, this name gives a way to find a vertex and working with it. This chapter shows some basic operations on graphs with named vertices.

- Check if there exists a vertex with a certain name: chapter 5.1

- Find a vertex by its name: chapter 5.2
- Get a named vertex its degree, in degree and out degree: chapter: 5.3
- Get a vertex its name from its vertex descriptor: chapter 5.4
- Set a vertex its name using its vertex descriptor: chapter 5.5
- Setting all vertices' names: chapter 5.6
- Clear a named vertex its edges: chapter 5.7
- Remove a named vertex: chapter 5.8
- Removing an edge between two named vertices: chapter 5.10
- Saving an directed/undirected graph with named vertices to a .dot file: chapter 5.15
- Loading a directed graph with named vertices from a .dot file: chapter 5.16
- Loading an undirected graph with named vertices from a .dot file: chapter 5.17

Especially the 'find_first_vertex_by_name' function (chapter 5.2) is important, as it shows how to obtain a vertex descriptor, which is used in later algorithms.

5.1 Check if there exists a vertex with a certain name

Before modifying our vertices, let's first determine if we can find a vertex by its name in a graph. After obtaining a name map, we obtain the vertex iterators, dereference these to obtain the vertex descriptors and then compare each vertex its name with the one desired.

Algorithm 77 Find if there is vertex with a certain name

```
#include <boost/graph/properties.hpp>

template <typename graph>
bool has_vertex_with_name(
    const std::string& vertex_name,
    const graph& g
) noexcept
{
    using vd = typename graph::vertex_descriptor;

    const auto vip = vertices(g);
    return std::find_if(vip.first, vip.second,
        [g, vertex_name](const vd& d)
        {
            const auto vertex_name_map
                = get(boost::vertex_name, g);
            return get(vertex_name_map, d) == vertex_name;
        }
    ) != vip.second;
}
```

This function can be demonstrated as in algorithm 78, where a certain name cannot be found in an empty graph. After adding the desired name, it is found.

Algorithm 78 Demonstration of the ‘has_vertex_with_name’ function

```
#include <cassert>

#include "add_named_vertex.h"
#include "create_empty_undirected_named_vertices_graph.h"
#include "has_vertex_with_name.h"

void has_vertex_with_name_demo() noexcept
{
    auto g
        = create_empty_undirected_named_vertices_graph();
    assert(!has_vertex_with_name("Felix", g));
    add_named_vertex("Felix", g);
    assert(has_vertex_with_name("Felix", g));
}
```

Note that this function only finds if there is at least one vertex with that

name: it does not tell how many vertices with that name exist in the graph.

5.2 Find a vertex by its name

Where STL functions work with iterators, here we obtain a vertex descriptor (see chapter 2.6) to obtain a handle to the desired vertex. Algorithm 79 shows how to obtain a vertex descriptor to the first (name) vertex found with a specific name.

Algorithm 79 Find the first vertex by its name

```
#include <cassert>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "has_vertex_with_name.h"

template <typename graph>
typename boost::graph_traits<graph>::vertex_descriptor
find_first_vertex_with_name(
    const std::string& name,
    const graph& g
) noexcept
{
    using vd = typename graph::vertex_descriptor;
    const auto vip = vertices(g);
    const auto i = std::find_if(
        vip.first, vip.second,
        [g, name](const vd d) {
            const auto vertex_name_map = get(boost::vertex_name, g);
            return get(vertex_name_map, d) == name;
        }
    );
    assert(i != vip.second);
    return *i;
}
```

With the vertex descriptor obtained, one can read and modify the vertex and the edges surrounding it. Algorithm 80 shows some examples of how to do so.

Algorithm 80 Demonstration of the ‘find_first_vertex_with_name’ function

```
#include <cassert>

#include "create_named_vertices_k2_graph.h"
#include "find_first_vertex_with_name.h"

void find_first_vertex_with_name_demo() noexcept
{
    const auto g
        = create_named_vertices_k2_graph();
    const auto vd
        = find_first_vertex_with_name(
            "My_computer", g
        );
    assert(
        out_degree(vd, g) == 1
    );
    assert(in_degree(vd, g) == 1);
}
```

5.3 Get a (named) vertex its degree, in degree and out degree

We already obtained all out degrees of all vertices in chapter 3.1 by just collecting all vertex descriptors. Here, we will search for a vertex with a certain name, obtain its vertex descriptor and find the number of connections it has.

With a vertex descriptor, we can read a vertex its types of degrees. Algorithm 79 shows how to find a vertex, obtain its vertex descriptor and then obtain the out degree from it.

Algorithm 81 Get the first vertex with a certain name its out degree from its vertex descriptor

```
#include <cassert>
#include "find_first_vertex_with_name.h"
#include "has_vertex_with_name.h"

template <typename graph>
int get_first_vertex_with_name_out_degree(
    const std::string& name,
    const graph& g) noexcept
{
    const auto vd
        = find_first_vertex_with_name(name, g);
    const int od {
        static_cast<int>(
            out_degree(vd, g)
        )
    };
    assert(static_cast<unsigned long>(od)
        == out_degree(vd, g)
    );
    return od;
}
```

Algorithm 82 shows how to use this function.

Algorithm 82 Demonstration of the ‘get_first_vertex_with_name_out_degree’ function

```
#include <cassert>
#include "create_named_vertices_k2_graph.h"
#include "get_first_vertex_with_name_out_degree.h"

void get_first_vertex_with_name_out_degree_demo()
    noexcept
{
    const auto g = create_named_vertices_k2_graph();
    assert(
        get_first_vertex_with_name_out_degree(
            "Me", g
        ) == 1
    );
    assert(
        get_first_vertex_with_name_out_degree(
            "My_computer", g
        ) == 1
    );
}
```

5.4 Get a vertex its name from its vertex descriptor

This may seem a trivial paragraph, as chapter 4.4 describes the ‘get_vertex_names’ algorithm, in which we get all vertices’ names. But it does not allow to first find a vertex of interest and subsequently getting only that one its name.

To obtain the name from a vertex descriptor, one needs to pull out the name map and then look up the vertex of interest.

Algorithm 83 Get a vertex its name from its vertex descriptor

```
#include <string>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>

template <typename graph>
std::string get_vertex_name(
    const typename boost::graph_traits<graph>::
        vertex_descriptor& vd,
    const graph& g
) noexcept
{
    const auto vertex_name_map
        = get(boost::vertex_name, g);
    return get(vertex_name_map, vd);
}
```

To use ‘get_vertex_name’, one first needs to obtain a vertex descriptor. Algorithm 84 shows a simple example:

Algorithm 84 Demonstration if the ‘get_vertex_name’ function

```
#include <cassert>

#include "add_named_vertex.h"
#include "create_empty_undirected_named_vertices_graph.h"
#include "find_first_vertex_with_name.h"
#include "get_vertex_name.h"

void get_vertex_name_demo() noexcept
{
    auto g
        = create_empty_undirected_named_vertices_graph();
    const std::string name{"Dex"};
    add_named_vertex(name, g);
    const auto vd
        = find_first_vertex_with_name(name, g);
    assert(get_vertex_name(vd, g) == name);
}
```

5.5 Set a (named) vertex its name from its vertex descriptor

If you know how to get the name from a vertex descriptor, setting it is just as easy, as shown in algorithm 85.

Algorithm 85 Set a vertex its name from its vertex descriptor

```
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>

template <typename graph>
void set_vertex_name(
    const std::string& any_vertex_name,
    const typename boost::graph_traits<graph>::
        vertex_descriptor& vd,
    graph& g
) noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const");

    auto vertex_name_map
        = get(boost::vertex_name, g);
    put(vertex_name_map, vd, any_vertex_name);
}
```

To use ‘set_vertex_name’, one first needs to obtain a vertex descriptor. Algorithm 86 shows a simple example.

Algorithm 86 Demonstration if the ‘set_vertex_name’ function

```
#include <cassert>

#include "add_named_vertex.h"
#include "create_empty_undirected_named_vertices_graph.h"
#include "find_first_vertex_with_name.h"
#include "get_vertex_name.h"
#include "set_vertex_name.h"

void set_vertex_name_demo() noexcept
{
    auto g
        = create_empty_undirected_named_vertices_graph();
    const std::string old_name{"Dex"};
    add_named_vertex(old_name, g);
    const auto vd
        = find_first_vertex_with_name(old_name, g);
    assert(get_vertex_name(vd, g) == old_name);
    const std::string new_name{"Diggy"};
    set_vertex_name(new_name, vd, g);
    assert(get_vertex_name(vd, g) == new_name);
}
```

5.6 Setting all vertices’ names

When the vertices of a graph have named vertices and you want to set all their names at once:

Algorithm 87 Setting the vertices' names

```
#include <string>
#include <vector>

#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>

template <typename graph>
void set_vertex_names(
    graph& g,
    const std::vector<std::string>& names
) noexcept
{
    static_assert(!std::is_const<graph>::value, "graph_
        cannot_be_const");

    const auto vertex_name_map
        = get(boost::vertex_name, g);
    auto ni = std::begin(names);
    const auto names_end = std::end(names);
    const auto vip = vertices(g);
    const auto j = vip.second;
    for (auto i = vip.first; i!=j; ++i, ++ni)
    {
        assert(ni != names_end);
        put(vertex_name_map, *i, *ni);
    }
}
```

A new function makes its appearance here: ‘put’ (not ‘boost::put’), which is the opposite of ‘get’ (not ‘boost::get’)

This is not a very usefull function if the graph is complex. But for just creating graphs for debugging, it may come in handy.

5.7 Clear the edges of a named vertex

A vertex descriptor can be used to clear all in/out/both edges connected to a vertex. It is necessary to remove these connections before the vertex itself can be removed. There are three functions to remove the edges connected to a vertex:

- `boost::clear_vertex`: removes all edges to and from the vertex
- `boost::clear_out_edges`: removes all outgoing edges from the vertex (in directed graphs only, else you will get a ‘error: no matching function for

call to `clear_out_edges`’, as described in chapter 24.2)

- `boost::clear_in_edges`: removes all incoming edges from the vertex (in directed graphs only, else you will get a ‘error: no matching function for call to `clear_in_edges`’, as described in chapter 24.3)

In the algorithm ‘`clear_first_vertex_with_name`’ the ‘`boost::clear_vertex`’ algorithm is used, as the graph used is undirectional:

Algorithm 88 Clear the first vertex with a certain name

```
#include <boost/graph/adjacency_list.hpp>
#include "find_first_vertex_with_name.h"
#include "has_vertex_with_name.h"

template <typename graph>
void clear_first_vertex_with_name(
    const std::string& name,
    graph& g
) noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const"
    );

    assert(has_vertex_with_name(name, g));
    const auto vd
        = find_first_vertex_with_name(name, g);
    boost::clear_vertex(vd, g);
}
```

Algorithm 89 shows the clearing of the first named vertex found.

Algorithm 89 Demonstration of the ‘`clear_first_vertex_with_name`’ function

```
#include <cassert>
#include "clear_first_vertex_with_name.h"
#include "create_named_vertices_k2_graph.h"

void clear_first_vertex_with_name_demo() noexcept
{
    auto g = create_named_vertices_k2_graph();
    assert(boost::num_edges(g) == 1);
    clear_first_vertex_with_name("My_computer", g);
    assert(boost::num_edges(g) == 0);
}
```

5.8 Remove a named vertex

A vertex descriptor can be used to remove a vertex from a graph. It is necessary to remove these connections (e.g. using ‘clear_first_vertex_with_name’, algorithm 88) before the vertex itself can be removed.

Removing a named vertex goes as follows: use the name of the vertex to get a first vertex descriptor, then call ‘boost::remove_vertex’, shown in algorithm 5.8:

Algorithm 90 Remove the first vertex with a certain name

```
#include <boost/graph/adjacency_list.hpp>
#include "find_first_vertex_with_name.h"
#include "has_vertex_with_name.h"

template <typename graph>
void remove_first_vertex_with_name(
    const std::string& name,
    graph& g
) noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const");

    assert(has_vertex_with_name(name, g));
    const auto vd
        = find_first_vertex_with_name(name, g);
    assert(degree(vd, g) == 0);
    boost::remove_vertex(vd, g);
}
```

Algorithm 91 shows the removal of the first named vertex found.

Algorithm 91 Demonstration of the ‘remove_first_vertex_with_name’ function

```
#include <cassert>

#include "clear_first_vertex_with_name.h"
#include "create_named_vertices_k2_graph.h"
#include "remove_first_vertex_with_name.h"

void remove_first_vertex_with_name_demo() noexcept
{
    auto g = create_named_vertices_k2_graph();
    clear_first_vertex_with_name(
        "My_computer", g
    );
    remove_first_vertex_with_name(
        "My_computer", g
    );
    assert(boost::num_edges(g) == 0);
    assert(boost::num_vertices(g) == 1);
}
```

Again, be sure that the vertex removed does not have any connections!

5.9 ► Adding an edge between two named vertices

Instead of looking for an edge descriptor, one can also add an edge from two vertex descriptors. Adding an edge between two named vertices named edge goes as follows: use the names of the vertices to get both vertex descriptors, then call ‘boost::add_edge’ on those two, as shown in algorithm 92.

Algorithm 92 Adding an edge between two named vertices

```
#include <cassert>
#include <string>
#include <boost/graph/adjacency_list.hpp>
#include "has_vertex_with_name.h"
#include "find_first_vertex_with_name.h"

template <typename graph>
typename boost::graph_traits<graph>::edge_descriptor
add_edge_between_named_vertices(
    const std::string& vertex_name_1,
    const std::string& vertex_name_2,
    graph& g
) noexcept
{
    assert(has_vertex_with_name(vertex_name_1, g));
    assert(has_vertex_with_name(vertex_name_2, g));
    const auto vd_1 = find_first_vertex_with_name(
        vertex_name_1, g);
    const auto vd_2 = find_first_vertex_with_name(
        vertex_name_2, g);
    const auto aer = boost::add_edge(vd_1, vd_2, g);
    assert(aer.second);
    return aer.first;
}
```

Algorithm 93 shows how to add an edge between two named vertices:

Algorithm 93 Demonstration of the 'add_edge_between_named_vertices' function

```
#include <cassert>
#include "add_edge_between_named_vertices.h"
#include "add_named_vertex.h"
#include "create_empty_undirected_named_vertices_graph.h"

void add_edge_between_named_vertices_demo() noexcept
{
    auto g = create_empty_undirected_named_vertices_graph();
    ;
    add_named_vertex("Bert", g);
    add_named_vertex("Ernie", g);
    add_edge_between_named_vertices("Bert", "Ernie", g);
    assert(boost::num_edges(g) == 1);
}
```

5.10 ► Removing the edge between two named vertices

Instead of looking for an edge descriptor, one can also remove an edge from two vertex descriptors (which is: the edge between the two vertices). Removing an edge between two named vertices named edge goes as follows: use the names of the vertices to get both vertex descriptors, then call 'boost::remove_edge' on those two, as shown in algorithm 94.

Algorithm 94 Remove the first edge with a certain name

```
#include "find_first_vertex_with_name.h"
#include "has_vertex_with_name.h"
#include "has_edge_between_vertices.h"

template <typename graph>
void remove_edge_between_vertices_with_names(
    const std::string& name_1,
    const std::string& name_2,
    graph& g
) noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const");

    assert(has_vertex_with_name(name_1, g));
    assert(has_vertex_with_name(name_2, g));
    const auto vd_1
        = find_first_vertex_with_name(name_1, g);
    const auto vd_2
        = find_first_vertex_with_name(name_2, g);
    assert(has_edge_between_vertices(vd_1, vd_2, g));
    boost::remove_edge(vd_1, vd_2, g);
}
```

Algorithm 95 shows the removal of the first named edge found.

Algorithm 95 Demonstration of the 'remove_edge_between_vertices_with_names' function

```
#include <cassert>

#include "create_named_edges_and_vertices_k3_graph.h"
#include "remove_edge_between_vertices_with_names.h"

void remove_edge_between_vertices_with_names_demo()
    noexcept
{
    auto g = create_named_edges_and_vertices_k3_graph();
    assert(boost::num_edges(g) == 3);
    remove_edge_between_vertices_with_names("top", "right", g);
    assert(boost::num_edges(g) == 2);
}
```

5.11 ► Count the vertices with a certain name

How often is a vertex with a certain name present? Here we'll find out.

Algorithm 96 Find the first vertex by its name

```
#include <string>
#include <boost/graph/properties.hpp>

template <typename graph>
int count_vertices_with_name(
    const std::string& name,
    const graph& g
) noexcept
{
    using vd = typename graph::vertex_descriptor;

    const auto vip = vertices(g);
    const auto cnt = std::count_if(
        vip.first, vip.second,
        [g, name](const vd& d)
        {
            const auto vertex_name_map
                = get(boost::vertex_name, g);
            return name
                == get(vertex_name_map, d);
        }
    );
    return static_cast<int>(cnt);
}
```

Here we use the STL `std::count_if` algorithm to count how many vertices have a name equal to the desired name.

Algorithm 97 shows some examples of how to do so.

Algorithm 97 Demonstration of the ‘find_first_vertex_with_name’ function

```
#include <cassert>
#include "add_named_vertex.h"
#include "count_vertices_with_name.h"
#include "create_empty_undirected_named_vertices_graph.h"
#include "create_named_vertices_path_graph.h"

void count_vertices_with_name_demo() noexcept
{
    auto g = create_named_vertices_path_graph(
        {"Apple", "Pear", "Apple"}
    );
    assert(count_vertices_with_name("Apple", g) == 2);
    assert(count_vertices_with_name("Pear", g) == 1);
    assert(count_vertices_with_name("Banana", g) == 0);
}
```

5.12 ► Create a direct-neighbour subgraph from a vertex descriptor of a graph with named vertices

Suppose you have a vertex of interest its vertex descriptor. Let’s say you want to get a subgraph of that vertex and its direct neighbours only. This means that all vertices of that subgraph are adjacent vertices and that the edges go either from focal vertex to its neighbours, or from adjacent vertex to adjacent neighbour.

Here is the ‘create_direct_neighbour_subgraph’ code:

Algorithm 98 Get the direct-neighbour named vertices subgraph from a vertex descriptor

```
#include <map>
#include <boost/graph/adjacency_list.hpp>
#include "get_vertex_name.h"
#include "set_vertex_name.h"
template <typename graph, typename vertex_descriptor>
graph create_direct_neighbour_named_vertices_subgraph(
    const vertex_descriptor& vd,
    const graph& g
)
{
    graph h;

    std::map<vertex_descriptor, vertex_descriptor> m;
    {
        const auto vd_h = boost::add_vertex(h);
        m.insert(std::make_pair(vd, vd_h));
    }
    //Copy vertices
    {
        const auto vdsi = boost::adjacent_vertices(vd, g);
        std::transform(vdsi.first, vdsi.second,
            std::inserter(m, std::begin(m)),
            [&h](const vertex_descriptor& d)
            {
                const auto vd_h = boost::add_vertex(h);
                return std::make_pair(d, vd_h);
            }
        );
    }
    //Copy edges
    {
        const auto eip = edges(g);
        const auto j = eip.second;
        for (auto i = eip.first; i!=j; ++i)
        {
            const auto vd_from = source(*i, g);
            const auto vd_to = target(*i, g);
            if (m.find(vd_from) == std::end(m)) continue;
            if (m.find(vd_to) == std::end(m)) continue;
            const auto aer = boost::add_edge(m[vd_from], m[vd_to], h);
            assert(aer.second);
        }
    }
    //Set vertex names
    for (const auto p: m) { 106
        set_vertex_name(get_vertex_name(p.first, g), p.second, h);
    }
    return h;
}
```

This demonstration code shows that the direct-neighbour graph of each vertex of a K_2 graphs is ... a K_2 graph!

Algorithm 99 Demo of the ‘create_direct_named_vertices_neighbour_subgraph’ function

```
#include "create_direct_neighbour_named_vertices_subgraph
.h"
#include "create_named_vertices_k2_graph.h"
#include "get_vertex_names.h"

void create_direct_neighbour_named_vertices_subgraph_demo
() noexcept
{
    const auto g = create_named_vertices_k2_graph();
    const auto vip = vertices(g);
    const auto j = vip.second;
    for (auto i=vip.first; i!=j; ++i) {
        const auto h =
            create_direct_neighbour_named_vertices_subgraph(
                *i,g
            );
        assert(boost::num_vertices(h) == 2);
        assert(boost::num_edges(h) == 1);
        const auto v = get_vertex_names(h);
        std::set<std::string> names(std::begin(v),std::end(v)
        );
        assert(names.count("Me") == 1);
        assert(names.count("My_computer") == 1);
    }
}
```

5.13 ► Creating all direct-neighbour subgraphs from a graph with named vertices

Using the previous function, it is easy to create all direct-neighbour subgraphs from a graph with named vertices:

Algorithm 100 Create all direct-neighbour subgraphs from a graph with named vertices

```
#include <vector>
#include "create_direct_neighbour_subgraph.h"

template <typename graph>
std::vector<graph>
    create_all_direct_neighbour_named_vertices_subgraphs(
        const graph g
    ) noexcept
{
    using vd = typename graph::vertex_descriptor;

    std::vector<graph> v;
    v.resize(boost::num_vertices(g));
    const auto vip = vertices(g);
    std::transform(
        vip.first, vip.second,
        std::begin(v),
        [g](const vd& d)
        {
            return create_direct_neighbour_subgraph(
                d, g
            );
        }
    );
    return v;
}
```

This demonstration code shows that all two direct-neighbour graphs of a K_2 graphs are ... K_2 graphs!

Algorithm 101 Demo of the ‘create_all_direct_neighbour_named_vertices_subgraphs’ function

```
#include <cassert>
#include "
    create_all_direct_neighbour_named_vertices_subgraphs.h
"
#include "create_named_vertices_k2_graph.h"

void
create_all_direct_neighbour_named_vertices_subgraphs_demo
() noexcept
{
    const auto v
    =
        create_all_direct_neighbour_named_vertices_subgraphs
        (create_named_vertices_k2_graph());
    assert(v.size() == 2);
    for (const auto g: v)
    {
        assert(boost::num_vertices(g) == 2);
        assert(boost::num_edges(g) == 1);
    }
}
```

5.14 ► Are two graphs with named vertices isomorphic?

Strictly speaking, finding isomorphisms is about the shape of the graph, independent of vertex name, and is already done in chapter 3.6.

Here, it is checked if two graphs with named vertices are ‘label isomorphic’ (please email me a better term if you know one). That is: if they have the same shape with the same vertex names at the same places.

To do this, there are two steps needed:

1. Map all vertex names to an unsigned int.
2. Compare the two graphs with that map

Below the class ‘named_vertex_invariant’ is shown. Its `std::map` maps the vertex names to an unsigned integer, which is done in the member function ‘collect_names’. The purpose of this, is that it is easier to compare integers than `std::strings`.

Algorithm 102 The named_vertex_invariant functor

```
#include <map>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/isomorphism.hpp>

template <class graph>
struct named_vertex_invariant {
    using str_to_int_map = std::map<std::string, size_t>;
    using result_type = size_t;
    using argument_type = typename graph::vertex_descriptor
        ;

    const graph& m_graph;
    str_to_int_map& m_mappings;

    size_t operator()(argument_type u) const {
        return m_mappings.at(boost::get(boost::vertex_name,
            m_graph, u));
    }
    size_t max() const noexcept { return m_mappings.size(); }

    void collect_names() noexcept {
        for (const auto vd : boost::make_iterator_range(boost::
            ::vertices(m_graph))) {
            const size_t next_id = m_mappings.size();
            const auto ins = m_mappings.insert(
                { boost::get(boost::vertex_name, m_graph, vd),
                    next_id }
            );
            if (ins.second) {
                //std::cout << "Mapped '" << ins.first->first <<
                "' to " << ins.first->second << "\n";
            }
        }
    }
};
```

To check for ‘label isomorphism’, multiple things need to be put in place for ‘boost::isomorphism’ to work with:

Algorithm 103 Check if two graphs with named vertices are isomorphic

```
#include "named_vertex_invariant.h"

#include <boost/graph/vf2_sub_graph_iso.hpp>
#include <boost/graph/graph_utility.hpp>

template <typename graph>
bool is_named_vertices_isomorphic(
    const graph &g,
    const graph &h
) noexcept {
    using vd = typename graph::vertex_descriptor;
    auto vertex_index_map = get(boost::vertex_index, g);
    std::vector<vd> iso(boost::num_vertices(g));

    typename named_vertex_invariant<graph>::str_to_int_map
        shared_names;
    named_vertex_invariant<graph> inv1{g, shared_names};
    named_vertex_invariant<graph> inv2{h, shared_names};
    inv1.collect_names();
    inv2.collect_names();

    return boost::isomorphism(g, h,
        boost::isomorphism_map(
            make_iterator_property_map(
                iso.begin(),
                vertex_index_map
            )
        )
        .vertex_invariant1(inv1)
        .vertex_invariant2(inv2)
    );
}
```

This demonstration code creates three path graphs, of which two are ‘label isomorphic’:

Algorithm 104 Demo of the ‘is_named_vertices_isomorphic’ function

```
#include <cassert>
#include "create_named_vertices_path_graph.h"
#include "is_named_vertices_isomorphic.h"

void is_named_vertices_isomorphic_demo() noexcept
{
    const auto g = create_named_vertices_path_graph(
        { "Alpha", "Beta", "Gamma" }
    );
    const auto h = create_named_vertices_path_graph(
        { "Gamma", "Beta", "Alpha" }
    );
    const auto i = create_named_vertices_path_graph(
        { "Alpha", "Gamma", "Beta" }
    );
    assert( is_named_vertices_isomorphic(g,h) );
    assert( !is_named_vertices_isomorphic(g,i) );
}
```

5.15 Saving an directed/undirected graph with named vertices to a .dot file

If you used the ‘create_named_vertices_k2_graph’ function (algorithm 65) to produce a K_2 graph with named vertices, you can store these names in multiple ways:

- Using `boost::make_label_writer`
- Using a C++11 lambda function

I show both ways, because you may need all of them.

The created .dot file is shown at algorithm 67.

You can use all characters in the vertex without problems (for example: comma’s, quotes, whitespace). This will not hold anymore for bundled and custom vertices in later chapters.

The ‘save_named_vertices_graph_to_dot’ functions below only save the structure of the graph and its vertex names. It ignores other edge and vertex properties.

5.15.1 Using `boost::make_label_writer`

The first implementation uses `boost::make_label_writer`, as shown in algorithm 105:

Algorithm 105 Saving a graph with named vertices to a .dot file

```
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include <boost/graph/properties.hpp>
#include "get_vertex_names.h"
#include "is_graphviz_friendly.h"

template <typename graph>
void save_named_vertices_graph_to_dot(
    const graph& g,
    const std::string& filename
) noexcept
{
    std::ofstream f(filename);
    const auto names
        = get_vertex_names(g); //Can be Graphviz-unfriendly
    boost::write_graphviz(
        f,
        g,
        boost::make_label_writer(&names[0])
    );
}
```

Here, the function `boost::write_graphviz` is called with a new, third argument. After collecting all names, these are used by `boost::make_label_writer` to write the names as labels.

5.15.2 Using a C++11 lambda function

An equivalent algorithm is algorithm 106:

Algorithm 106 Saving a graph with named vertices to a .dot file using a lambda expression

```
#include <string>
#include <ostream>
#include <boost/graph/graphviz.hpp>
#include <boost/graph/properties.hpp>

#include "get_vertex_names.h"

template <typename graph>
void save_named_vertices_graph_to_dot_using_lambda(
    const graph& g,
    const std::string& filename
) noexcept
{
    using vd_t = typename graph::vertex_descriptor;
    std::ofstream f(filename);
    const auto name_map = get(boost::vertex_name, g);
    boost::write_graphviz(
        f,
        g,
        [name_map](std::ostream& os, const vd_t& vd) {
            const std::string s{name_map[vd]};
            if (s.find(' ') == std::string::npos) {
                //No space, no quotes around string
                os << "[label=" << s << " ]";
            }
            else {
                //Has space, put quotes around string
                os << "[label=\"" << s << "\" ]";
            }
        }
    );
}
```

In this code, a lambda function is used as a third argument.

A lambda function is an on-the-fly function that has these parts:

- the capture brackets '[]', to take variables within the lambda function
- the function argument parentheses '()', to put the function arguments in
- the function body '{}', where to write what it does

First we create a shorthand for the vertex descriptor type, that we'll need to use a lambda function argument (in C++14 you can use auto).

We then create a vertex name map at function scope (in C++17 this can be at lambda function scope) and pass it to the lambda function using its capture section.

The lambda function arguments need to be two: a `std::ostream&` (a reference to a general out-stream) and a vertex descriptor. In the function body, we get the name of the vertex the same as the ‘`get_vertex_name`’ function (algorithm 83) and stream it to the out stream.

5.15.3 Demonstration

Algorithm 107 shows how to use (one of) the ‘`save_named_vertices_graph_to_dot`’ function(s):

Algorithm 107 Demonstration of the ‘`save_named_vertices_graph_to_dot`’ function

```
#include "create_named_vertices_k2_graph.h"
#include "create_named_vertices_markov_chain.h"
#include "save_named_vertices_graph_to_dot.h"

void save_named_vertices_graph_to_dot_demo() noexcept
{
    const auto g = create_named_vertices_k2_graph();
    save_named_vertices_graph_to_dot(
        g, "create_named_vertices_k2_graph.dot"
    );

    const auto h = create_named_vertices_markov_chain();
    save_named_vertices_graph_to_dot(
        h, "create_named_vertices_markov_chain.dot"
    );
}
```

When using the ‘`save_named_vertices_graph_to_dot`’ function (algorithm 105), only the structure of the graph and the vertex names are saved: all other properties like edge name are not stored. Algorithm 151 shows how to do so.

5.16 Loading a directed graph with named vertices from a .dot

When loading a graph from file, one needs to specify a type of graph. In this example, an directed graph with named vertices is loaded, as shown in algorithm 108:

Algorithm 108 Loading a directed graph with named vertices from a .dot file

```
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "create_empty_directed_named_vertices_graph.h"
#include "is_regular_file.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    boost::property<
        boost::vertex_name_t, std::string
    >
>
>
load_directed_named_vertices_graph_from_dot(
    const std::string& dot_filename
)
{
    assert(is_regular_file(dot_filename));
    std::ifstream f(dot_filename.c_str());
    auto g = create_empty_directed_named_vertices_graph();
    boost::dynamic_properties dp(boost::
        ignore_other_properties);
    dp.property("label", get(boost::vertex_name, g));
    boost::read_graphviz(f, g, dp);
    return g;
}
```

In this algorithm, first it is checked if the file to load exists. Then an empty directed graph is created. Next to this, a `boost::dynamic_properties` is created with its default constructor, after which we direct the `boost::dynamic_properties` to find a ‘node_id’ and ‘label’ in the vertex name map. From this and the empty graph, ‘`boost::read_graphviz`’ is called to build up the graph.

Algorithm 109 shows how to use the ‘`load_directed_graph_from_dot`’ function:

Algorithm 109 Demonstration of the ‘load_directed_named_vertices_graph_from_dot’ function

```
#include "create_named_vertices_markov_chain.h"
#include "load_directed_named_vertices_graph_from_dot.h"
#include "save_named_vertices_graph_to_dot.h"
#include "get_vertex_names.h"

void load_directed_named_vertices_graph_from_dot_demo()
    noexcept
{
    using boost::num_edges;
    using boost::num_vertices;

    const auto g
        = create_named_vertices_markov_chain();
    const std::string filename{
        "create_named_vertices_markov_chain.dot"
    };
    save_named_vertices_graph_to_dot(g, filename);
    const auto h
        = load_directed_named_vertices_graph_from_dot(
            filename
        );
    assert(num_edges(g) == num_edges(h));
    assert(num_vertices(g) == num_vertices(h));
    assert(get_vertex_names(g) == get_vertex_names(h));
}
```

This demonstration shows how the Markov chain is created using the ‘create_named_vertices_markov_chain’ function (algorithm 21), saved and then loaded. The loaded graph is checked to be a directed graph similar to the Markov chain with the same vertex names (using the ‘get_vertex_names’ function, algorithm 60).

5.17 Loading an undirected graph with named vertices from a .dot

When loading a graph from file, one needs to specify a type of graph. In this example, an undirected graph with named vertices is loaded, as shown in algorithm 110:

Algorithm 110 Loading an undirected graph with named vertices from a .dot file

```
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "create_empty_undirected_named_vertices_graph.h"
#include "is_regular_file.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
        boost::vertex_name_t, std::string
    >
>
>
load_undirected_named_vertices_graph_from_dot(
    const std::string& dot_filename
)
{
    assert(is_regular_file(dot_filename));
    std::ifstream f(dot_filename.c_str());
    auto g = create_empty_undirected_named_vertices_graph();
    ;
    boost::dynamic_properties dp(boost::
        ignore_other_properties);
    dp.property("label", get(boost::vertex_name, g));
    boost::read_graphviz(f, g, dp);
    return g;
}
```

The only difference with loading a directed graph, is that the initial empty graph is undirected instead. Chapter 5.16 describes the rationale of this function.

Algorithm 111 shows how to use the ‘load_undirected_graph_from_dot’ function:

Algorithm 111 Demonstration of the ‘load_undirected_graph_from_dot’ function

```
#include "create_named_vertices_k2_graph.h"
#include "load_undirected_named_vertices_graph_from_dot.h"
"

#include "save_named_vertices_graph_to_dot.h"
#include "get_vertex_names.h"

void load_undirected_named_vertices_graph_from_dot_demo()
    noexcept
{
    using boost::num_edges;
    using boost::num_vertices;

    const auto g
        = create_named_vertices_k2_graph();
    const std::string filename{
        "create_named_vertices_k2_graph.dot"
    };
    save_named_vertices_graph_to_dot(g, filename);
    const auto h
        = load_undirected_named_vertices_graph_from_dot(
            filename
        );
    assert(num_edges(g) == num_edges(h));
    assert(num_vertices(g) == num_vertices(h));
    assert(get_vertex_names(g) == get_vertex_names(h));
}
```

This demonstration shows how K_2 with named vertices is created using the ‘create_named_vertices_k2_graph’ function (algorithm 65), saved and then loaded. The loaded graph is checked to be an undirected graph similar to K_2 , with the same vertex names (using the ‘get_vertex_names’ function, algorithm 60).

6 Building graphs with named edges and vertices

Up until now, the graphs created have had edges and vertices without any property. In this chapter, graphs will be created, in which edges vertices can have a name. This name will be of the `std::string` data type, but other types are possible as well. There are many more built-in properties edges and nodes can have (see the `boost/graph/properties.hpp` file for these).

In this chapter, we will build the following graphs:

- An empty directed graph that allows for edges and vertices with names: see chapter 6.1
- An empty undirected graph that allows for edges and vertices with names: see chapter 6.2
- Markov chain with named edges and vertices: see chapter 6.6
- K_3 with named edges and vertices: see chapter 6.8

In the process, some basic (sometimes bordering trivial) functions are shown:

- Adding an named edge: see chapter 6.3
- Getting the edges' names: see chapter 6.5

These functions are mostly there for completion and showing which data types are used.

6.1 Creating an empty directed graph with named edges and vertices

Let's create a trivial empty directed graph, in which the both the edges and vertices can have a name:

Algorithm 112 Creating an empty directed graph with named edges and vertices

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

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    boost::property<boost::vertex_name_t, std::string>,
    boost::property<boost::edge_name_t, std::string>
>
create_empty_directed_named_edges_and_vertices_graph()
    noexcept
{
    return {};
}
```

This graph:

- has its out edges stored in a `std::vector` (due to the first `boost::vecS`)
- has its vertices stored in a `std::vector` (due to the second `boost::vecS`)

- is directed (due to the `boost::directedS`)
- The vertices have one property: they have a name, that is of data type `std::string` (due to the `boost::property< boost::vertex_name_t, std::string>`)
- The edges have one property: they have a name, that is of data type `std::string` (due to the `boost::property< boost::edge_name_t, std::string>`)
- The graph has no properties
- Edges are stored in a `std::list`

The `boost::adjacency_list` has a new, fifth template argument '`boost::property< boost::edge_name_t, std::string>`'. This can be read as: "edges have the property '`boost::edge_name_t`', that is of data type '`std::string`'". Or simply: "edges have a name that is stored as a `std::string`".

Algorithm 113 shows how to create this graph. Note that all the earlier functions defined in this tutorial keep working as expected.

| | | | | | |
|------------------|------------|---------------|----|-----|---|
| Algorithm | 113 | Demonstration | if | the | 'cre- |
| | | | | | ate_empty_directed_named_edges_and_vertices_graph' function |

```
#include <cassert>
#include "add_named_edge.h"
#include "
    create_empty_directed_named_edges_and_vertices_graph.h
"
#include "get_edge_names.h"
#include "get_vertex_names.h"

void
create_empty_directed_named_edges_and_vertices_graph_demo
() noexcept
{
    using strings = std::vector<std::string>;
    auto g
        =
        create_empty_directed_named_edges_and_vertices_graph
        ();
    add_named_edge("Reed", g);
    const strings expected_vertex_names{"", ""};
    const strings vertex_names = get_vertex_names(g);
    assert(expected_vertex_names == vertex_names);
    const strings expected_edge_names{"Reed"};
    const strings edge_names = get_edge_names(g);
    assert(expected_edge_names == edge_names);
}
```

6.2 Creating an empty undirected graph with named edges and vertices

Let's create a trivial empty undirected graph, in which the both the edges and vertices can have a name:

Algorithm 114 Creating an empty undirected graph with named edges and vertices

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

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<boost::vertex_name_t, std::string>,
    boost::property<boost::edge_name_t, std::string>
>
create_empty_undirected_named_edges_and_vertices_graph()
    noexcept
{
    return {};
}
```

This graph:

- has its out edges stored in a `std::vector` (due to the first `boost::vecS`)
- has its vertices stored in a `std::vector` (due to the second `boost::vecS`)
- is undirected (due to the `boost::undirectedS`)
- The vertices have one property: they have a name, that is of data type `std::string` (due to the `boost::property< boost::vertex_name_t, std::string>`)
- The edges have one property: they have a name, that is of data type `std::string` (due to the `boost::property< boost::edge_name_t, std::string>`)
- The graph has no properties
- Edges are stored in a `std::list`

The `boost::adjacency_list` has a new, fifth template argument '`boost::property< boost::edge_name_t, std::string>`'. This can be read as: "edges have the property '`boost::edge_name_t`', that is of data type '`std::string`'". Or simply: "edges have a name that is stored as a `std::string`".

Algorithm 115 shows how to create this graph. Note that all the earlier functions defined in this tutorial keep working as expected.

Algorithm 115 Demonstration if the ‘create_empty_undirected_named_edges_and_vertices_graph’ function

```
#include <cassert>

#include "add_named_edge.h"
#include "
    create_empty_undirected_named_edges_and_vertices_graph
    .h"
#include "get_edge_names.h"
#include "get_vertex_names.h"

void
create_empty_undirected_named_edges_and_vertices_graph_demo
() noexcept
{
    using strings = std::vector<std::string>;
    auto g
        =
        create_empty_undirected_named_edges_and_vertices_graph
        ();
    add_named_edge("Reed", g);
    const strings expected_vertex_names{"", ""};
    const strings vertex_names = get_vertex_names(g);
    assert(expected_vertex_names == vertex_names);
    const strings expected_edge_names{"Reed"};
    const strings edge_names = get_edge_names(g);
    assert(expected_edge_names == edge_names);
}
```

6.3 Adding a named edge

Adding an edge with a name:

Algorithm 116 Add a vertex with a name

```
#include <cassert>
#include <string>
#include <boost/graph/adjacency_list.hpp>

template <typename graph>
typename boost::graph_traits<graph>::edge_descriptor
add_named_edge(
    const std::string& edge_name,
    graph& g
) noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const");
    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);
    const auto aer = boost::add_edge(vd_a, vd_b, g);
    assert(aer.second);

    auto edge_name_map = get(
        boost::edge_name, g
    );
    put(edge_name_map, aer.first, edge_name);
    return aer.first;
}
```

In this code snippet, the edge descriptor (see chapter 2.12 if you need to refresh your memory) when using ‘boost::add_edge’ is used as a key to change the edge its name map.

The algorithm 117 shows how to add a named edge to an empty graph. When trying to add named vertices to graph without this property, you will get the error ‘formed reference to void’ (see chapter 24.1).

Algorithm 117 Demonstration of the ‘add_named_edge’ function

```
#include <cassert>
#include "add_named_edge.h"
#include "
    create_empty_undirected_named_edges_and_vertices_graph
    .h"

void add_named_edge_demo() noexcept
{
    auto g
        =
        create_empty_undirected_named_edges_and_vertices_graph
        ();
    add_named_edge("Richards", g);
    assert(boost::num_edges(g) == 1);
}
```

6.4 Adding a named edge between vertices

When having two vertex descriptors, you can add a named edge between those.

Algorithm 118 Add a vertex with a name between vertices

```
#include <cassert>
#include <string>
#include <boost/graph/adjacency_list.hpp>

template <typename graph, typename vertex_descriptor>
typename boost::graph_traits<graph>::edge_descriptor
add_named_edge_between_vertices(
    const std::string& edge_name,
    const vertex_descriptor from,
    const vertex_descriptor to,
    graph& g
) noexcept
{
    const auto aer = boost::add_edge(from, to, g);
    assert(aer.second);

    auto edge_name_map = get(
        boost::edge_name, g
    );
    put(edge_name_map, aer.first, edge_name);

    return aer.first;
}
```

In this code snippet, the edge is added between the two vertex descriptors, after which the name of the edge is set.

A demonstration is given by algorithm 119:

Algorithm 119 Demonstration of the ‘add_named_edge_between_vertices’ function

```
#include <cassert>
#include "add_named_edge.h"
#include "
    create_empty_undirected_named_edges_and_vertices_graph
    .h"

void add_named_edge_demo() noexcept
{
    auto g
        =
            create_empty_undirected_named_edges_and_vertices_graph
            ();
    add_named_edge("Richards", g);
    assert(boost::num_edges(g) == 1);
}
```

6.5 Getting the edges’ names

When the edges of a graph have named vertices, one can extract them as such:

Algorithm 120 Get the edges' names

```
#include <string>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/properties.hpp>

template <typename graph>
std::vector<std::string> get_edge_names(const graph& g)
    noexcept
{
    using boost::graph_traits;
    using ed = typename graph_traits<graph>::
        edge_descriptor;
    std::vector<std::string> v(boost::num_edges(g));
    const auto eip = edges(g);
    std::transform(eip.first, eip.second, std::begin(v),
        [g](const ed& d)
        {
            const auto edge_name_map = get(boost::edge_name, g);
            return get(edge_name_map, d);
        }
    );
    return v;
}
```

The names of the edges are obtained from a `boost::property_map` and then put into a `std::vector`. The algorithm 121 shows how to apply this function.

The order of the edge names may be different after saving and loading.

Would you dare to try to get the edges' names from a graph without vertices with names, you will get the error 'formed reference to void' (see chapter 24.1).

Algorithm 121 Demonstration of the ‘get_edge_names’ function

```
#include <cassert>

#include "add_named_edge.h"
#include "
    create_empty_undirected_named_edges_and_vertices_graph
    .h"
#include "get_edge_names.h"

void get_edge_names_demo() noexcept
{
    auto g
        =
            create_empty_undirected_named_edges_and_vertices_graph
            ();
    const std::string edge_name_1{"Eugene"};
    const std::string edge_name_2{"Another_Eugene"};
    add_named_edge(edge_name_1, g);
    add_named_edge(edge_name_2, g);
    const std::vector<std::string> expected_names{
        edge_name_1, edge_name_2
    };
    const std::vector<std::string> edge_names{
        get_edge_names(g)
    };
    assert(expected_names == edge_names);
}
```

6.6 Creating Markov chain with named edges and vertices

6.6.1 Graph

We build this graph:

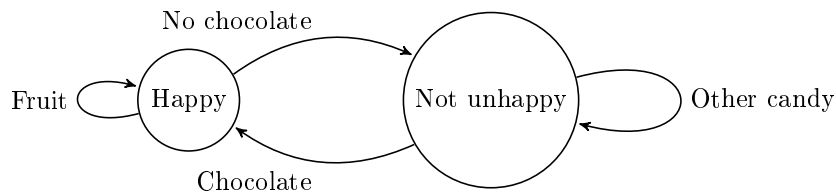


Figure 26: A two-state Markov chain where the edges and vertices have texts

6.6.2 Function to create such a graph

Here is the code:

Algorithm 122 Creating the two-state Markov chain as depicted in figure 26

```
#include <string>
#include "
    create_empty_directed_named_edges_and_vertices_graph.h
#include "add_named_vertex.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    boost::property<boost::vertex_name_t, std::string>,
    boost::property<boost::edge_name_t, std::string>
>
create_named_edges_and_vertices_markov_chain() noexcept
{
    auto g
        =
            create_empty_directed_named_edges_and_vertices_graph
            ();
    const auto vd_a = add_named_vertex("Happy", g);
    const auto vd_b = add_named_vertex("Not_unhappy", g);
    const auto aer_aa = boost::add_edge(vd_a, vd_a, g);
    assert(aer_aa.second);
    const auto aer_ab = boost::add_edge(vd_a, vd_b, g);
    assert(aer_ab.second);
    const auto aer_ba = boost::add_edge(vd_b, vd_a, g);
    assert(aer_ba.second);
    const auto aer_bb = boost::add_edge(vd_b, vd_b, g);
    assert(aer_bb.second);

    auto edge_name_map = get(
        boost::edge_name, g
    );
    put(edge_name_map, aer_aa.first, "Fruit");
    put(edge_name_map, aer_ab.first, "No_chocolate");
    put(edge_name_map, aer_ba.first, "Chocolate");
    put(edge_name_map, aer_bb.first, "Other_candy");

    return g;
}
```

6.6.3 Creating such a graph

Here is the demo:

Algorithm 123 Demo of the ‘create_named_edges_and_vertices_markov_chain’ function (algorithm 122)

```
#include <cassert>
#include <iostream>
#include "create_named_edges_and_vertices_markov_chain.h"
#include "get_edge_names.h"
#include "get_vertex_names.h"

void create_named_edges_and_vertices_markov_chain_demo()
    noexcept
{
    using strings = std::vector<std::string>;

    const auto g
        = create_named_edges_and_vertices_markov_chain();

    const strings expected_vertex_names{
        "Happy", "Not_unhappy"
    };
    const strings vertex_names{
        get_vertex_names(g)
    };
    assert(expected_vertex_names == vertex_names);

    const strings expected_edge_names{
        "Fruit", "No_chocolate", "Chocolate", "Other_candy"
    };

    const strings edge_names{get_edge_names(g)};
    assert(expected_edge_names == edge_names);
}
```

6.6.4 The .dot file produced

Algorithm 124 .dot file created from the 'create_named_edges_and_vertices_markov_chain' function (algorithm 122), converted from graph to .dot file using algorithm 48

```
digraph G {
0[label=Happy];
1[label="Not unhappy"];
0->0 [label="Fruit"];
0->1 [label="No chocolate"];
1->0 [label="Chocolate"];
1->1 [label="Other candy"];
}
```

6.6.5 The .svg file produced

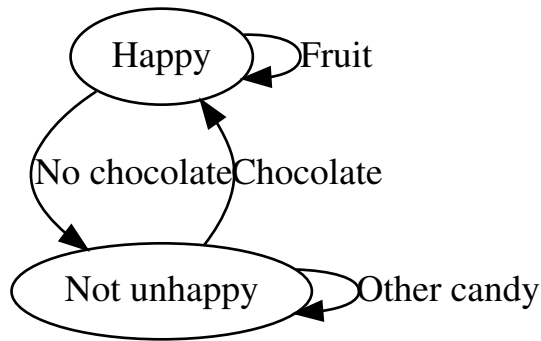


Figure 27: .svg file created from the 'create_named_edges_and_vertices_markov_chain' function (algorithm 122) its .dot file, converted from .dot file to .svg using algorithm 339

6.7 Creating K_2 with named edges and vertices

6.7.1 Graph

We extend the graph K_2 with named vertices of chapter 4.6 by adding names to the edges, as depicted in figure 28:

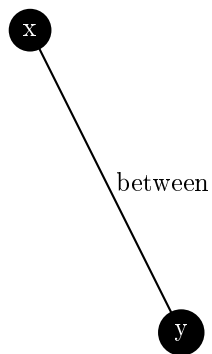


Figure 28: K_2 : a fully connected graph with three named edges and vertices

6.7.2 Function to create such a graph

To create K_2 , the following code can be used:

Algorithm 125 Creating K_2 as depicted in figure 28

```
#include <string>
#include <boost/graph/adjacency_list.hpp>
#include "
    create_empty_undirected_named_edges_and_vertices_graph
    .h"
#include "add_named_vertex.h"
#include "add_named_edge_between_vertices.h"
boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<boost::vertex_name_t, std::string>,
    boost::property<boost::edge_name_t, std::string>
>
create_named_edges_and_vertices_k2_graph() noexcept
{
    auto g
        =
            create_empty_undirected_named_edges_and_vertices_graph
            ();
    const auto vd_a = add_named_vertex("x", g);
    const auto vd_b = add_named_vertex("y", g);
    add_named_edge_between_vertices("between", vd_a, vd_b, g)
        ;
    return g;
}
```

Most of the code is a repeat of algorithm 65. In the end, the edge names are obtained as a `boost::property_map` and set.

6.7.3 Creating such a graph

Algorithm 126 shows how to create the graph and measure its edge and vertex names.

Algorithm 126 Demonstration of the ‘create_named_edges_and_vertices_k2’ function

```

#include <cassert>
#include <iostream>
#include "create_named_edges_and_vertices_k2_graph.h"
#include "get_edge_names.h"
#include "get_vertex_names.h"

void create_named_edges_and_vertices_k2_graph_demo()
    noexcept
{
    using strings = std::vector<std::string>;

    const auto g
        = create_named_edges_and_vertices_k2_graph();

    const strings expected_vertex_names{
        "x", "y"
    };
    const strings vertex_names{
        get_vertex_names(g)
    };
    assert(expected_vertex_names == vertex_names);

    const strings expected_edge_names{
        "between"
    };
    const strings edge_names{get_edge_names(g)};
    assert(expected_edge_names == edge_names);
}

```

6.7.4 The .dot file produced

Algorithm 127 .dot file created from the ‘create_named_edges_and_vertices_k2_graph’ function (algorithm 125), converted from graph to .dot file using algorithm 48

```

graph G {
0[label=x];
1[label=y];
0--1 [label="between"];
}

```

6.7.5 The .svg file produced

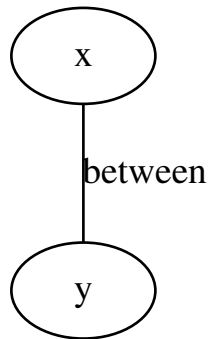


Figure 29: .svg file created from the ‘create_named_edges_and_vertices_k2_graph’ function (algorithm 125) its .dot file, converted from .dot file to .svg using algorithm 339

6.8 Creating K_3 with named edges and vertices

6.8.1 Graph

We extend the graph K_2 with named vertices of chapter 4.6 by adding names to the edges, as depicted in figure 30:

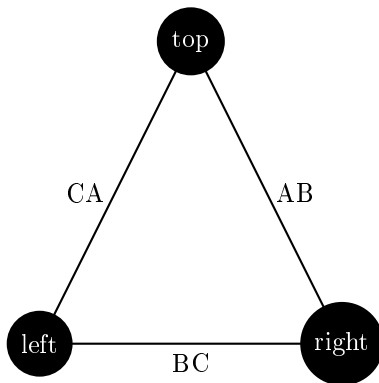


Figure 30: K_3 : a fully connected graph with three named edges and vertices

6.8.2 Function to create such a graph

To create K_3 , the following code can be used:

Algorithm 128 Creating K_3 as depicted in figure 30

```
#include <string>
#include <boost/graph/adjacency_list.hpp>
#include "
    create_empty_undirected_named_edges_and_vertices_graph
    .h"
#include "add_named_vertex.h"
#include "add_named_edge_between_vertices.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<boost::vertex_name_t, std::string>,
    boost::property<boost::edge_name_t, std::string>
>
create_named_edges_and_vertices_k3_graph() noexcept
{
    auto g
        =
            create_empty_undirected_named_edges_and_vertices_graph
            ();
    const auto vd_a = add_named_vertex("top", g);
    const auto vd_b = add_named_vertex("right", g);
    const auto vd_c = add_named_vertex("left", g);
    add_named_edge_between_vertices("AB", vd_a, vd_b, g);
    add_named_edge_between_vertices("BC", vd_b, vd_c, g);
    add_named_edge_between_vertices("CA", vd_c, vd_a, g);
    return g;
}
```

Most of the code is a repeat of algorithm 65. In the end, the edge names are obtained as a `boost::property_map` and `set`.

6.8.3 Creating such a graph

Algorithm 129 shows how to create the graph and measure its edge and vertex names.

Algorithm 129 Demonstration of the ‘create_named_edges_and_vertices_k3’ function

```
#include <cassert>
#include <iostream>
#include "create_named_edges_and_vertices_k3_graph.h"
#include "get_edge_names.h"
#include "get_vertex_names.h"

void create_named_edges_and_vertices_k3_graph_demo()
    noexcept
{
    using strings = std::vector<std::string>;

    const auto g
        = create_named_edges_and_vertices_k3_graph();

    const strings expected_vertex_names{
        "top", "right", "left"
    };
    const strings vertex_names{
        get_vertex_names(g)
    };
    assert(expected_vertex_names == vertex_names);

    const strings expected_edge_names{
        "AB", "BC", "CA"
    };
    const strings edge_names{get_edge_names(g)};
    assert(expected_edge_names == edge_names);
}
```

6.8.4 The .dot file produced

Algorithm 130 .dot file created from the ‘create_named_edges_and_vertices_k3_graph’ function (algorithm 128), converted from graph to .dot file using algorithm 48

```
graph G {  
0[label=top];  
1[label=right];  
2[label=left];  
0--1 [label="AB"];  
1--2 [label="BC"];  
2--0 [label="CA"];  
}
```

6.8.5 The .svg file produced

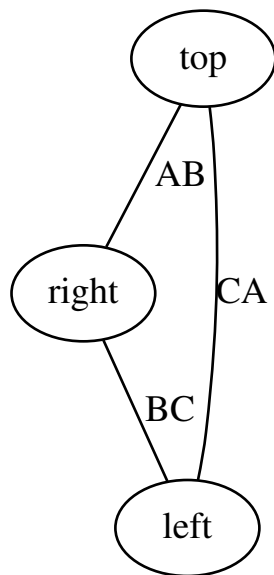


Figure 31: .svg file created from the ‘create_named_edges_and_vertices_k3_graph’ function (algorithm 128) its .dot file, converted from .dot file to .svg using algorithm 339

6.9 ► Creating a path graph with named edges and vertices

Here we create a path graph with names edges and vertices

6.9.1 Graph

Here I show a path graph with four vertices (see figure 32):

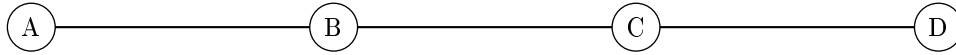


Figure 32: A path graph with four vertices

6.9.2 Function to create such a graph

To create a path graph, the following code can be used:

Algorithm 131 Creating a path graph as depicted in figure 32

```
#include <vector>
#include "add_named_edge_between_vertices.h"
#include "add_named_vertex.h"
#include "
    create_empty_undirected_named_edges_and_vertices_graph
    .h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<boost::vertex_name_t, std::string>,
    boost::property<boost::edge_name_t, std::string>
>
create_named_edges_and_vertices_path_graph(
    const std::vector<std::string>& edge_names,
    const std::vector<std::string>& vertex_names
) noexcept
{
    assert(vertex_names.empty()
        || vertex_names.size() == edge_names.size() + 1
    );
    auto g =
        create_empty_undirected_named_edges_and_vertices_graph
        ();
    if (vertex_names.size() == 0) { return g; }
    auto vd_1 = add_named_vertex(*vertex_names.begin(), g);
    if (vertex_names.size() == 1) return g;
    const auto j = std::end(vertex_names);
    auto vertex_name = std::begin(vertex_names);
    auto edge_name = std::begin(edge_names);
    for (++vertex_name; vertex_name != j; ++vertex_name, ++
        edge_name) //Skip first vertex name
    {
        auto vd_2 = add_named_vertex(*vertex_name, g);
        add_named_edge_between_vertices(
            *edge_name, vd_1, vd_2, g
        );
        vd_1 = vd_2;
    }
    return g;
}
```

6.9.3 Creating such a graph

Algorithm 132 demonstrates how to create a path graph with named edges and vertices and checks if it has the correct amount of edges and vertices:

Algorithm 132 Demonstration of ‘create_named_edges_and_vertices_path_graph’

```
#include <cassert>
#include "create_named_edges_and_vertices_path_graph.h"
#include "get_edge_names.h"
#include "get_vertex_names.h"

void create_named_edges_and_vertices_path_graph_demo()
    noexcept
{
    const std::vector<std::string> vertex_names
        = {"A", "B", "C", "D"};
    const std::vector<std::string> edge_names
        = { "1", "2", "3" };
    const auto g =
        create_named_edges_and_vertices_path_graph(
            edge_names, vertex_names
        );
    assert(boost::num_edges(g) == 3);
    assert(boost::num_vertices(g) == 4);
    assert(get_edge_names(g) == edge_names);
    assert(get_vertex_names(g) == vertex_names);
}
```

6.9.4 The .dot file produced

This graph can be converted to the .dot file as shown in algorithm 133:

Algorithm 133 .dot file created from the ‘create_named_edges_and_vertices_path_graph’ function (algorithm 131), converted from graph to .dot file using algorithm 48

6.9.5 The .svg file produced

The .dot file can be converted to the .svg as shown in figure 33:

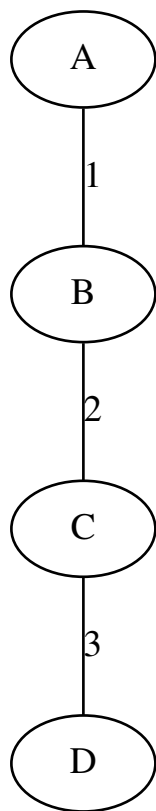


Figure 33: .svg file created from the ‘create_named_edges_and_vertices_path_graph’ function (algorithm 131) its .dot file, converted from .dot file to .svg using algorithm 339

6.10 ► Creating a Petersen graph with named edges and vertices

Here we create a Petersen graph with named edges and vertices.

6.10.1 Graph

Here I show a Petersen graph (see figure 34):

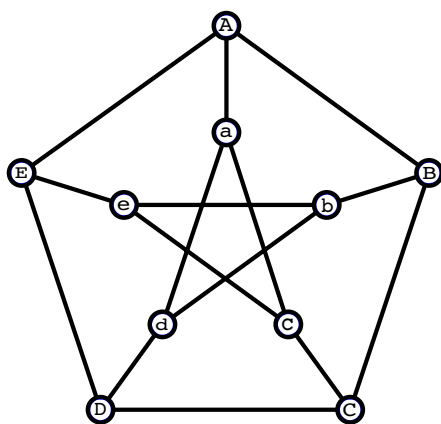


Figure 34: A Petersen graph with named edges and vertices (modified from https://en.wikipedia.org/wiki/Petersen_graph)

6.10.2 Function to create such a graph

To create a Petersen graph with named edges and vertices, the following code can be used:

Algorithm 134 Creating a Petersen graph as depicted in figure 34

```
#include <cassert>
#include <vector>
#include "add_named_vertex.h"
#include "create_empty_undirected_named_vertices_graph.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
        boost::vertex_name_t, std::string
    >
>
>
create_named_vertices_petersen_graph() noexcept
{
    auto g = create_empty_undirected_named_vertices_graph()
        ;
    using vd = decltype(
        create_empty_undirected_named_vertices_graph())::
        vertex_descriptor;

    std::vector<vd> v; //Outer
    for (int i=0; i!=5; ++i) {
        v.push_back(
            add_named_vertex(std::string(1, 'A' + i), g)
        );
    }
    std::vector<vd> w; //Inner
    for (int i=0; i!=5; ++i) {
        w.push_back(
            add_named_vertex(std::string(1, 'a' + i), g)
        );
    }
    //Outer ring
    for (int i=0; i!=5; ++i) {
        const auto aer
            = boost::add_edge(v[i], v[(i + 1) % 5], g);
        assert(aer.second);
    }
    //Spoke
    for (int i=0; i!=5; ++i) {
        const auto aer
            = boost::add_edge(v[i], w[i], g);
        assert(aer.second);
    }
    //Inner pentagram
    for (int i=0; i!=5; ++i) {
        const auto aer
            = boost::add_edge(w[i], w[(i + 2) % 5], g);
        assert(aer.second);
    }
    return g;
}
```

6.10.3 Creating such a graph

Algorithm 135 demonstrates how to create a path graph with named vertices and checks if it has the correct amount of edges and vertices:

Algorithm 135 Demonstration of ‘create_named_vertices_petersen_graph’

```
#include <cassert>
#include "create_named_vertices_petersen_graph.h"

void create_named_vertices_petersen_graph_demo() noexcept
{
    const auto g = create_named_vertices_petersen_graph();
    assert(boost::num_edges(g) == 15);
    assert(boost::num_vertices(g) == 10);
}
```

6.10.4 The .dot file produced

This graph can be converted to the .dot file as shown in algorithm 136:

Algorithm 136 .dot file created from the ‘create_named_edges_and_vertices_petersen_graph’ function (algorithm 134), converted from graph to .dot file using algorithm 48

```
graph G {
0[label=A];
1[label=B];
2[label=C];
3[label=D];
4[label=E];
5[label=a];
6[label=b];
7[label=c];
8[label=d];
9[label=e];
0--1 [label="F"];
1--2 [label="G"];
2--3 [label="H"];
3--4 [label="I"];
4--0 [label="J"];
0--5 [label="O"];
1--6 [label="1"];
2--7 [label="2"];
3--8 [label="3"];
4--9 [label="4"];
5--7 [label="f"];
6--8 [label="g"];
7--9 [label="h"];
8--5 [label="i"];
9--6 [label="j"];
}
```

6.10.5 The .svg file produced

The .dot file can be converted to the .svg as shown in figure 35:

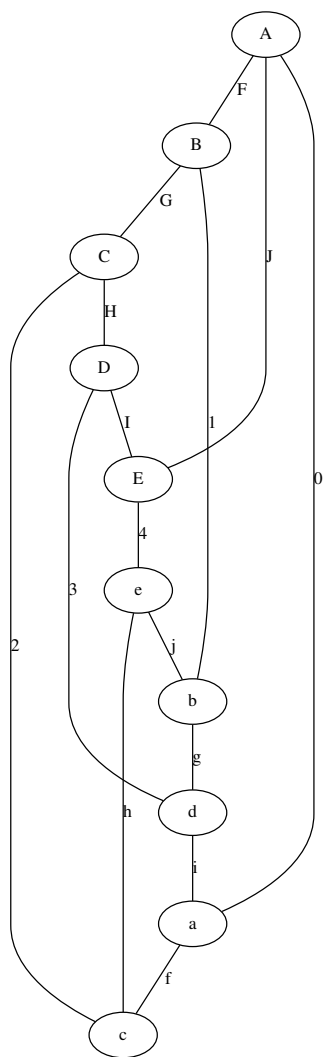


Figure 35: .svg file created from the ‘create_named_edges_and_vertices_petersen_graph’ function (algorithm 134) its .dot file, converted from .dot file to .svg using algorithm 339

7 Working on graphs with named edges and vertices

Working with named edges...

- Check if there exists an edge with a certain name: chapter 7.1

- Find a (named) edge by its name: chapter 7.2
- Get a (named) edge its name from its edge descriptor: chapter 7.3
- Set a (named) edge its name using its edge descriptor: chapter 7.4
- Remove a named edge: chapter 7.5
- Saving a graph with named edges and vertices to a .dot file: chapter 7.8
- Loading a directed graph with named edges and vertices from a .dot file: chapter 7.9
- Loading an undirected graph with named edges and vertices from a .dot file: chapter 7.10

Especially chapter 7.2 with the ‘find_first_edge_by_name’ algorithm shows how to obtain an edge descriptor, which is used in later algorithms.

7.1 Check if there exists an edge with a certain name

Before modifying our edges, let’s first determine if we can find an edge by its name in a graph. After obtaining a name map, we obtain the edge iterators, dereference these to obtain the edge descriptors and then compare each edge its name with the one desired.

Algorithm 137 Find if there is an edge with a certain name

```
#include <string>
#include <boost/graph/properties.hpp>

template <typename graph>
bool has_edge_with_name(
    const std::string& edge_name,
    const graph& g
) noexcept
{
    using ed = typename boost::graph_traits<graph>::
        edge_descriptor;
    const auto eip = edges(g);
    return std::find_if(eip.first, eip.second,
        [edge_name, g](const ed& d)
        {
            const auto edge_name_map
                = get(boost::edge_name, g);
            return get(edge_name_map, d) == edge_name;
        }) != eip.second;
}
```

This function can be demonstrated as in algorithm 138, where a certain name cannot be found in an empty graph. After adding the desired name, it is found.

Algorithm 138 Demonstration of the ‘has_edge_with_name’ function

```
#include <cassert>

#include "add_named_edge.h"
#include "
    create_empty_undirected_named_edges_and_vertices_graph
    .h"
#include "has_edge_with_name.h"

void has_edge_with_name_demo() noexcept
{
    auto g =
        create_empty_undirected_named_edges_and_vertices_graph
        ();
    assert (!has_edge_with_name("Edward", g));
    add_named_edge("Edward", g);
    assert (has_edge_with_name("Edward", g));
}
```

Note that this function only finds if there is at least one edge with that name: it does not tell how many edges with that name exist in the graph.

7.2 Find an edge by its name

Where STL functions work with iterators, here we obtain an edge descriptor (see chapter 2.12) to obtain a handle to the desired edge. Algorithm 139 shows how to obtain an edge descriptor to the first (name) edge found with a specific name.

Algorithm 139 Find the first edge by its name

```
#include <string>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>

template <typename graph>
typename boost::graph_traits<graph>::edge_descriptor
find_first_edge_with_name(
    const std::string& name,
    const graph& g
) noexcept
{
    using ed = typename boost::graph_traits<graph>::
        edge_descriptor;
    const auto eip = edges(g);
    const auto i = std::find_if(
        eip.first, eip.second,
        [g, name](const ed d) {
            const auto edge_name_map = get(boost::edge_name, g)
                ;
            return get(edge_name_map, d) == name;
        }
    );
    assert(i != eip.second);
    return *i;
}
```

With the edge descriptor obtained, one can read and modify the graph. Algorithm 140 shows some examples of how to do so.

Algorithm 140 Demonstration of the ‘find_first_edge_by_name’ function

```
#include <cassert>

#include "create_named_edges_and_vertices_k3_graph.h"
#include "find_first_edge_with_name.h"

void find_first_edge_with_name_demo() noexcept
{
    const auto g
        = create_named_edges_and_vertices_k3_graph();
    const auto ed
        = find_first_edge_with_name("AB", g);
    assert(boost::source(ed,g) != boost::target(ed,g));
}
```

7.3 Get a (named) edge its name from its edge descriptor

This may seem a trivial paragraph, as chapter 6.5 describes the ‘get_edge_names’ algorithm, in which we get all edges’ names. But it does not allow to first find an edge of interest and subsequently getting only that one its name.

To obtain the name from an edgedescriptor, one needs to pull out the name map and then look up the edge of interest.

Algorithm 141 Get an edge its name from its edge descriptor

```
#include <string>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>

template <typename graph>
std::string get_edge_name(
    const typename boost::graph_traits<graph>::
        edge_descriptor& ed,
    const graph& g
) noexcept
{
    const auto edge_name_map
        = get(boost::edge_name,
            g
        );
    return get(edge_name_map, ed);
}
```

To use ‘get_edge_name’, one first needs to obtain an edge descriptor. Al-

gorithm 142 shows a simple example.

Algorithm 142 Demonstration if the ‘get_edge_name’ function

```
#include <cassert>

#include "add_named_edge.h"
#include "
    create_empty_undirected_named_edges_and_vertices_graph
    .h"
#include "find_first_edge_with_name.h"
#include "get_edge_name.h"

void get_edge_name_demo() noexcept
{
    auto g =
        create_empty_undirected_named_edges_and_vertices_graph
        ();
    const std::string name{"Dex"};
    add_named_edge(name, g);
    const auto ed = find_first_edge_with_name(name, g);
    assert(get_edge_name(ed, g) == name);
}
```

7.4 Set a (named) edge its name from its edge descriptor

If you know how to get the name from an edge descriptor, setting it is just as easy, as shown in algorithm 143.

Algorithm 143 Set an edge its name from its edge descriptor

```
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>

template <typename graph>
void set_edge_name(
    const std::string& any_edge_name,
    const typename boost::graph_traits<graph>::
        edge_descriptor& vd,
    graph& g
) noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const");

    auto edge_name_map = get(boost::edge_name, g);
    put(edge_name_map, vd, any_edge_name);
}
```

To use ‘set_edge_name’, one first needs to obtain an edge descriptor. Algorithm 144 shows a simple example.

Algorithm 144 Demonstration if the ‘set_edge_name’ function

```
#include <cassert>

#include "add_named_edge.h"
#include "
    create_empty_undirected_named_edges_and_vertices_graph
    .h"
#include "find_first_edge_with_name.h"
#include "get_edge_name.h"
#include "set_edge_name.h"

void set_edge_name_demo() noexcept
{
    auto g =
        create_empty_undirected_named_edges_and_vertices_graph
        ();
    const std::string old_name{"Dex"};
    add_named_edge(old_name, g);
    const auto vd = find_first_edge_with_name(old_name, g);
    assert(get_edge_name(vd, g) == old_name);
    const std::string new_name{"Diggy"};
    set_edge_name(new_name, vd, g);
    assert(get_edge_name(vd, g) == new_name);
}
```

7.5 Removing the first edge with a certain name

An edge descriptor can be used to remove an edge from a graph.

Removing a named edge goes as follows: use the name of the edge to get a first edge descriptor, then call ‘boost::remove_edge’, shown in algorithm 90:

Algorithm 145 Remove the first edge with a certain name

```
#include <boost/graph/adjacency_list.hpp>
#include "find_first_edge_with_name.h"
#include "has_edge_with_name.h"

template <typename graph>
void remove_first_edge_with_name(
    const std::string& name,
    graph& g
) noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const");

    assert(has_edge_with_name(name, g));
    const auto vd
        = find_first_edge_with_name(name, g);
    boost::remove_edge(vd, g);
}
```

Algorithm 146 shows the removal of the first named edge found.

Algorithm 146 Demonstration of the ‘remove_first_edge_with_name’ function

```
#include <cassert>

#include "create_named_edges_and_vertices_k3_graph.h"
#include "remove_first_edge_with_name.h"

void remove_first_edge_with_name_demo() noexcept
{
    auto g = create_named_edges_and_vertices_k3_graph();
    assert(boost::num_edges(g) == 3);
    assert(boost::num_vertices(g) == 3);
    remove_first_edge_with_name("AB", g);
    assert(boost::num_edges(g) == 2);
    assert(boost::num_vertices(g) == 3);
}
```

7.6 ► Create a direct-neighbour subgraph from a vertex descriptor of a graph with named edges and vertices

Suppose you have a vertex of interest its vertex descriptor. Let's say you want to get a subgraph of that vertex and its direct neighbours only. This means that all vertices of that subgraph are adjacent vertices and that the edges go either from focal vertex to its neighbours, or from adjacent vertex to adjacent neighbour.

Here is the 'create_direct_neighbour_subgraph' code:

Algorithm 147 Get the direct-neighbour named edges and vertices subgraph from a vertex descriptor

```

#include <map>
#include <boost/graph/adjacency_list.hpp>
#include "add_named_edge_between_vertices.h"
#include "add_named_vertex.h"
#include "get_edge_name.h"
#include "get_vertex_name.h"
// #include "set_edge_name.h"
// #include "set_vertex_name.h"
template <typename graph, typename vertex_descriptor>
graph
    create_direct_neighbour_named_edges_and_vertices_subgraph
    (
        const vertex_descriptor& vd,
        const graph& g
    )
{
    graph h;

    std::map<vertex_descriptor, vertex_descriptor> vds;
    {
        const auto vd_h = add_named_vertex(get_vertex_name(vd, g), h);
        vds.insert(std::make_pair(vd, vd_h));
    }
    // Copy vertices
    {
        const auto vdsi = boost::adjacent_vertices(vd, g);
        std::transform(vdsi.first, vdsi.second,
            std::inserter(vds, std::begin(vds)),
            [g, &h](const vertex_descriptor& d)
            {
                const auto vd_h = add_named_vertex(
                    get_vertex_name(d, g), h);
                return std::make_pair(d, vd_h);
            }
        );
    }
    // Copy edges
    {
        const auto eip = edges(g);
        const auto j = eip.second;
        for (auto i = eip.first; i != j; ++i)
        {
            const auto vd_from = source(*i, g);
            const auto vd_to = target(*i, g);
            if (vds.find(vd_from) == std::end(vds)) continue;
            if (vds.find(vd_to) == std::end(vds)) continue;
            add_named_edge_between_vertices(
                get_edge_name(*i, g),
                vds[vd_from], vds[vd_to], h
            );
        }
    }
}

```

This demonstration code shows that the direct-neighbour graph of each vertex of a K_2 graphs is ... a K_2 graph!

Algorithm 148 Demo of the ‘create_direct_neighbour_named_edges_and_vertices_subgraph’ function

```
#include "
    create_direct_neighbour_named_edges_and_vertices_subgraph
    .h"
#include "create_named_edges_and_vertices_k2_graph.h"
#include "get_edge_names.h"
#include "get_vertex_names.h"

void
    create_direct_neighbour_named_edges_and_vertices_subgraph_demo
    () noexcept
{
    const auto g = create_named_edges_and_vertices_k2_graph
        ();
    const auto vip = vertices(g);
    const auto j = vip.second;
    for (auto i=vip.first; i!=j; ++i) {
        const auto h =
            create_direct_neighbour_named_edges_and_vertices_subgraph
            (
                *i,g
            );
        assert(boost::num_vertices(h) == 2);
        assert(boost::num_edges(h) == 1);
        const auto v = get_vertex_names(h);
        std::set<std::string> vs(std::begin(v),std::end(v));
        assert(vs.count("x") == 1);
        assert(vs.count("y") == 1);
        const auto e = get_edge_names(h);
        std::set<std::string> es(std::begin(e),std::end(e));
        assert(es.count("between") == 1);
    }
}
```

7.7 ► Creating all direct-neighbour subgraphs from a graph with named edges and vertices

Using the previous function, it is easy to create all direct-neighbour subgraphs from a graph with named edges and vertices:

Algorithm 149 Create all direct-neighbour subgraphs from a graph with named edges and vertices

```
#include <vector>
#include "create_direct_neighbour_subgraph.h"

template <typename graph>
std::vector<graph>
    create_all_direct_neighbour_named_edges_and_vertices_subgraphs
    (
        const graph g
    ) noexcept
{
    using vd = typename graph::vertex_descriptor;

    std::vector<graph> v;
    v.resize(boost::num_vertices(g));
    const auto vip = vertices(g);
    std::transform(
        vip.first, vip.second,
        std::begin(v),
        [g](const vd& d)
        {
            return create_direct_neighbour_subgraph(
                d, g
            );
        }
    );
    return v;
}
```

This demonstration code shows that all two direct-neighbour graphs of a K_2 graphs are ... K_2 graphs!

Algorithm 150 Demo of the ‘create_all_direct_neighbour_named_edges_and_vertices_subgraphs’ function

```
#include <cassert>
#include "
    create_all_direct_neighbour_named_edges_and_vertices_subgraphs
    .h"
#include "create_named_edges_and_vertices_k2_graph.h"

void
    create_all_direct_neighbour_named_edges_and_vertices_subgraphs_demo
    () noexcept
{
    const auto v
        =
            create_all_direct_neighbour_named_edges_and_vertices_subgraphs
            (create_named_edges_and_vertices_k2_graph());
    assert(v.size() == 2);
    for (const auto g: v)
    {
        assert(boost::num_vertices(g) == 2);
        assert(boost::num_edges(g) == 1);
    }
}
```

7.8 Saving an undirected graph with named edges and vertices as a .dot

If you used the create_named_edges_and_vertices_k3_graph function (algorithm 128) to produce a K_3 graph with named edges and vertices, you can store these names additionally with algorithm 151:

Algorithm 151 Saving an undirected graph with named edges and vertices to a .dot file

```
#include <string>
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include <boost/graph/properties.hpp>

#include "get_edge_names.h"
#include "get_vertex_names.h"

template <typename graph>
void save_named_edges_and_vertices_graph_to_dot(
    const graph& g,
    const std::string& filename
)
{
    using my_edge_descriptor = typename graph::
        edge_descriptor;

    std::ofstream f(filename);
    const auto vertex_names = get_vertex_names(g);
    const auto edge_name_map = boost::get(boost::edge_name,
        g);
    boost::write_graphviz(
        f,
        g,
        boost::make_label_writer(&vertex_names[0]),
        [edge_name_map](std::ostream& out, const
            my_edge_descriptor& e) {
            out << "[label=\"" << edge_name_map[e] << "\"]";
        }
    );
}
```

If you created a graph with edges more complex than just a name, you will still just write these to the .dot file. Chapter 13.7 shows how to write custom vertices to a .dot file.

So, the ‘save_named_edges_and_vertices_graph_to_dot’ function (algorithm 48) saves only the structure of the graph and its edge and vertex names.

7.9 Loading a directed graph with named edges and vertices from a .dot

When loading a graph from file, one needs to specify a type of graph. In this example, an directed graph with named edges and vertices is loaded, as shown in algorithm 152:

Algorithm 152 Loading a directed graph with named edges and vertices from a .dot file

```
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "
    create_empty_directed_named_edges_and_vertices_graph.h
"
#include "is_regular_file.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    boost::property<
        boost::vertex_name_t, std::string
    >,
    boost::property<
        boost::edge_name_t, std::string
    >
>
>
load_directed_named_edges_and_vertices_graph_from_dot(
    const std::string& dot_filename
)
{
    assert(is_regular_file(dot_filename));
    std::ifstream f(dot_filename.c_str());
    auto g =
        create_empty_directed_named_edges_and_vertices_graph
        ();
    boost::dynamic_properties dp(boost::
        ignore_other_properties);
    dp.property("label", get(boost::vertex_name, g));
    dp.property("edge_id", get(boost::edge_name, g));
    dp.property("label", get(boost::edge_name, g));
    boost::read_graphviz(f, g, dp);
    return g;
}
```

In this algorithm, first it is checked if the file to load exists. Then an empty directed graph is created. Next to this, a `boost::dynamic_properties` is created with its default constructor, after which we direct the `boost::dynamic_properties` to find a ‘node_id’ and ‘label’ in the vertex name map, ‘edge_id’ and ‘label’ to the edge name map. From this and the empty graph, ‘`boost::read_graphviz`’ is called to build up the graph.

Algorithm 153 shows how to use the ‘`load_directed_graph_from_dot`’ function:

Algorithm 153 Demonstration of the ‘`load_directed_named_edges_and_vertices_graph_from_dot`’ function

```
#include "create_named_edges_and_vertices_markov_chain.h"
#include "
    load_directed_named_edges_and_vertices_graph_from_dot.
    h"
#include "save_named_edges_and_vertices_graph_to_dot.h"
#include "get_vertex_names.h"

void
load_directed_named_edges_and_vertices_graph_from_dot_demo
() noexcept
{
    using boost::num_edges;
    using boost::num_vertices;

    const auto g
        = create_named_edges_and_vertices_markov_chain();
    const std::string filename{
        "create_named_edges_and_vertices_markov_chain.dot"
    };
    save_named_edges_and_vertices_graph_to_dot(g, filename)
        ;
    const auto h
        =
            load_directed_named_edges_and_vertices_graph_from_dot
            (
                filename
            );
    assert(num_edges(g) == num_edges(h));
    assert(num_vertices(g) == num_vertices(h));
    assert(get_vertex_names(g) == get_vertex_names(h));
}
```

This demonstration shows how the Markov chain is created using the ‘cre-

ate_named_edges_and_vertices_markov_chain' function (algorithm 122), saved and then loaded. The loaded graph is checked to be a directed graph similar to the Markov chain with the same edge and vertex names (using the 'get_edge_names' function, algorithm 120, and the 'get_vertex_names' function, algorithm 60).

7.10 Loading an undirected graph with named edges and vertices from a .dot

When loading a graph from file, one needs to specify a type of graph. In this example, an undirected graph with named edges and vertices is loaded, as shown in algorithm 154:

Algorithm 154 Loading an undirected graph with named edges and vertices from a .dot file

```
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "
    create_empty_undirected_named_edges_and_vertices_graph
    .h"
#include "is_regular_file.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
        boost::vertex_name_t, std::string
    >,
    boost::property<
        boost::edge_name_t, std::string
    >
>
>
load_undirected_named_edges_and_vertices_graph_from_dot(
    const std::string& dot_filename
)
{
    assert(is_regular_file(dot_filename));
    std::ifstream f(dot_filename.c_str());
    auto g =
        create_empty_undirected_named_edges_and_vertices_graph
        ();
    boost::dynamic_properties dp(boost::
        ignore_other_properties);
    dp.property("label", get(boost::vertex_name, g));
    dp.property("edge_id", get(boost::edge_name, g));
    dp.property("label", get(boost::edge_name, g));
    boost::read_graphviz(f, g, dp);
    return g;
}
```

The only difference with loading a directed graph, is that the initial empty graph is undirected instead. Chapter 7.9 describes the rationale of this function.

Algorithm 155 shows how to use the 'load_undirected_graph_from_dot' function:

Algorithm 155 Demonstration of the ‘load_undirected_named_edges_and_vertices_graph_from_dot’ function

```

#include "create_named_edges_and_vertices_k3_graph.h"
#include "
    load_undirected_named_edges_and_vertices_graph_from_dot
    .h"
#include "save_named_edges_and_vertices_graph_to_dot.h"
#include "get_vertex_names.h"

void
    load_undirected_named_edges_and_vertices_graph_from_dot_demo
    () noexcept
{
    using boost::num_edges;
    using boost::num_vertices;

    const auto g
        = create_named_edges_and_vertices_k3_graph();
    const std::string filename{
        "create_named_edges_and_vertices_k3_graph.dot"
    };
    save_named_edges_and_vertices_graph_to_dot(g, filename)
        ;
    const auto h
        =
            load_undirected_named_edges_and_vertices_graph_from_dot
            (
                filename
            );
    assert(num_edges(g) == num_edges(h));
    assert(num_vertices(g) == num_vertices(h));
    assert(get_vertex_names(g) == get_vertex_names(h));
}

```

This demonstration shows how K_3 with named edges and vertices is created using the ‘create_named_edges_and_vertices_k3_graph’ function (algorithm 128), saved and then loaded. The loaded graph is checked to be an undirected graph similar to K_3 , with the same edge and vertex names (using the ‘get_edge_names’ function, algorithm 120, and the ‘get_vertex_names’ function, algorithm 60).

8 Building graphs with bundled vertices

Up until now, the graphs created have had edges and vertices with the built-in name property. In this chapter, graphs will be created, in which the vertices can have a bundled ‘my_bundled_vertex’ type⁷. The following graphs will be created:

- An empty directed graph that allows for bundled vertices: see chapter 157
- An empty undirected graph that allows for bundled vertices: see chapter 8.2
- A two-state Markov chain with bundled vertices: see chapter 8.6
- K_2 with bundled vertices: see chapter 8.7

In the process, some basic (sometimes bordering trivial) functions are shown:

- Create the vertex class, called ‘my_bundled_vertex’: see chapter 8.1
- Adding a ‘my_bundled_vertex’: see chapter 8.4
- Getting the vertices ‘my_bundled_vertex’-es: see chapter 8.5

These functions are mostly there for completion and showing which data types are used.

8.1 Creating the bundled vertex class

Before creating an empty graph with bundled vertices, that bundled vertex class must be created. In this tutorial, it is called ‘my_bundled_vertex’. ‘my_bundled_vertex’ is a class that is nonsensical, but it can be replaced by any other class type.

Here I will show the header file of ‘my_bundled_vertex’, as the implementation of it is not important:

⁷I do not intend to be original in naming my data types

Algorithm 156 Declaration of `my_bundled_vertex`

```
#include <string>
#include <iosfwd>
#include <boost/property_map/dynamic_property_map.hpp>

struct my_bundled_vertex
{
    explicit my_bundled_vertex(
        const std::string& name = "",
        const std::string& description = "",
        const double x = 0.0,
        const double y = 0.0
    ) noexcept;
    std::string m_name;
    std::string m_description;
    double m_x;
    double m_y;
};

bool operator==(const my_bundled_vertex& lhs, const
    my_bundled_vertex& rhs) noexcept;
bool operator!=(const my_bundled_vertex& lhs, const
    my_bundled_vertex& rhs) noexcept;
```

‘`my_bundled_vertex`’ is a class that has multiple properties:

- It has four public member variables: the double ‘`m_x`’ (‘`m_`’ stands for member), the double ‘`m_y`’, the `std::string m_name` and the `std::string m_description`. These variables must be public
- It has a default constructor
- It is copyable
- It is comparable for equality (it has `operator==`), which is needed for searching

‘`my_bundled_vertex`’ does not have to have the stream operators defined for file I/O, as this goes via the public member variables.

8.2 Create the empty directed graph with bundled vertices

Algorithm 157 Creating an empty directed graph with bundled vertices

```
#include <boost/graph/adjacency_list.hpp>
#include "my_bundled_vertex.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    my_bundled_vertex
>
create_empty_directed_bundled_vertices_graph() noexcept
{
    return {};
}
```

This graph:

- has its out edges stored in a `std::vector` (due to the first `boost::vecS`)
- has its vertices stored in a `std::vector` (due to the second `boost::vecS`)
- is directed (due to the `boost::directedS`)
- The vertices have one property: they have a bundled type, that is of data type ‘`my_bundled_vertex`’
- The edges and graph have no properties
- Edges are stored in a `std::list`

The `boost::adjacency_list` has a new, fourth template argument ‘`my_bundled_vertex`’. This can be read as: “vertices have the bundled property ‘`my_bundled_vertex`’”. Or simply: “vertices have a bundled type called `my_bundled_vertex`”.

8.3 Create the empty undirected graph with bundled vertices

Algorithm 158 Creating an empty undirected graph with bundled vertices

```
#include <boost/graph/adjacency_list.hpp>
#include "my_bundled_vertex.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    my_bundled_vertex
>
create_empty_undirected_bundled_vertices_graph() noexcept
{
    return {};
}
```

This code is very similar to the code described in chapter 8.2, except that the directedness (the third template argument) is undirected (due to the `boost::undirectedS`).

8.4 Add a bundled vertex

Adding a bundled vertex is very similar to adding a named vertex (chapter 4.3).

Algorithm 159 Add a bundled vertex

```
#include <boost/graph/adjacency_list.hpp>
#include "my_bundled_vertex.h"

template <typename graph>
typename boost::graph_traits<graph>::vertex_descriptor
add_bundled_vertex(const my_bundled_vertex& v, graph& g)
    noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const");

    const auto vd = boost::add_vertex(g);
    g[vd] = v;
    return vd;
}
```

When having added a new (abstract) vertex to the graph, the vertex descriptor is used to set the ‘my_bundled_vertex’ in the graph.

8.5 Getting the bundled vertices’ my_vertexes⁸

When the vertices of a graph have any bundled ‘my_bundled_vertex’, one can extract these as such:

Algorithm 160 Get the bundled vertices’ my_vertexes

```
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "my_bundled_vertex.h"

template <typename graph>
std::vector<my_bundled_vertex> get_my_bundled_vertexes(
    const graph& g
) noexcept
{
    using vd = typename graph::vertex_descriptor;

    std::vector<my_bundled_vertex> v(boost::num_vertices(g)
    );
    const auto vip = vertices(g);
    std::transform(vip.first, vip.second, std::begin(v),
        [g](const vd& d) { return g[d]; }
    );
    return v;
}
```

The ‘my_bundled_vertex’ bundled in each vertex is obtained from a vertex descriptor and then put into a std::vector.

The order of the ‘my_bundled_vertex’ objects may be different after saving and loading.

When trying to get the vertices’ my_bundled_vertex from a graph without these, you will get the error ‘formed reference to void’ (see chapter 24.1).

⁸the name ‘my_vertexes’ is chosen to indicate this function returns a container of my_vertex

8.6 Creating a two-state Markov chain with bundled vertices

8.6.1 Graph

Figure 36 shows the graph that will be reproduced:

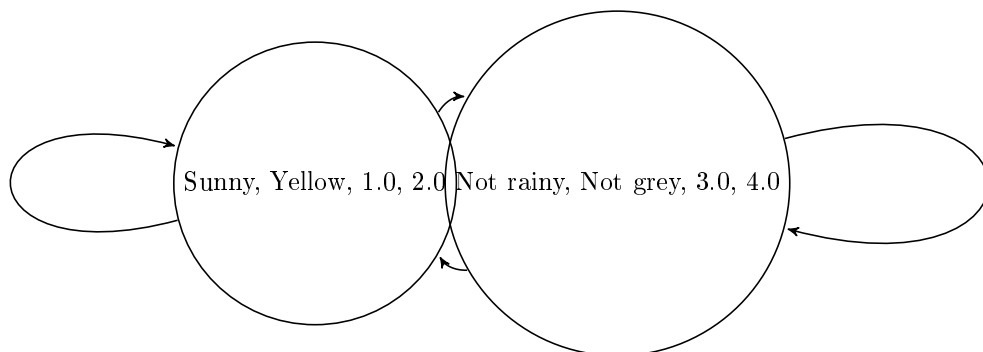


Figure 36: A two-state Markov chain where the vertices have bundled properties and the edges have no properties. The vertices' properties are nonsensical

8.6.2 Function to create such a graph

Here is the code creating a two-state Markov chain with bundled vertices:

Algorithm 161 Creating the two-state Markov chain as depicted in figure 36

```
#include <cassert>
#include "create_empty_directed_bundled_vertices_graph.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    my_bundled_vertex
>
create_bundled_vertices_markov_chain() noexcept
{
    auto g
        = create_empty_directed_bundled_vertices_graph();
    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);
    const auto aer_aa = boost::add_edge(vd_a, vd_a, g);
    assert(aer_aa.second);
    const auto aer_ab = boost::add_edge(vd_a, vd_b, g);
    assert(aer_ab.second);
    const auto aer_ba = boost::add_edge(vd_b, vd_a, g);
    assert(aer_ba.second);
    const auto aer_bb = boost::add_edge(vd_b, vd_b, g);
    assert(aer_bb.second);

    g[vd_a] = my_bundled_vertex("Sunny",
        "Yellow", 1.0, 2.0
    );
    g[vd_b] = my_bundled_vertex("Not_rainy",
        "Not_grey", 3.0, 4.0
    );

    return g;
}
```

8.6.3 Creating such a graph

Here is the demo:

Algorithm 162 Demo of the ‘create_bundled_vertices_markov_chain’ function (algorithm 161)

```
#include <cassert>
#include "create_bundled_vertices_markov_chain.h"
#include "get_my_bundled_vertexes.h"
#include "get_my_bundled_vertex.h"

void create_bundled_vertices_markov_chain_demo() noexcept
{
    const auto g
        = create_bundled_vertices_markov_chain();
    const std::vector<my_bundled_vertex> expected{
        my_bundled_vertex("Sunny","Yellow",1.0,2.0),
        my_bundled_vertex("Not_rainy","Not_grey",3.0,4.0)
    };
    const auto found = get_my_bundled_vertexes(g);
    assert(expected == found);
}
```

8.6.4 The .dot file produced

Algorithm 163 .dot file created from the ‘create_bundled_vertices_markov_chain’ function (algorithm 161), converted from graph to .dot file using algorithm 176

```
digraph G {
0[label="Sunny",comment="Yellow",width=1,height=2];
1[label="Not$$$SPACE$$$rainy",comment="Not$$$SPACE$$$grey",width=3,height=4];
0->0 ;
0->1 ;
1->0 ;
1->1 ;
}
```

8.6.5 The .svg file produced

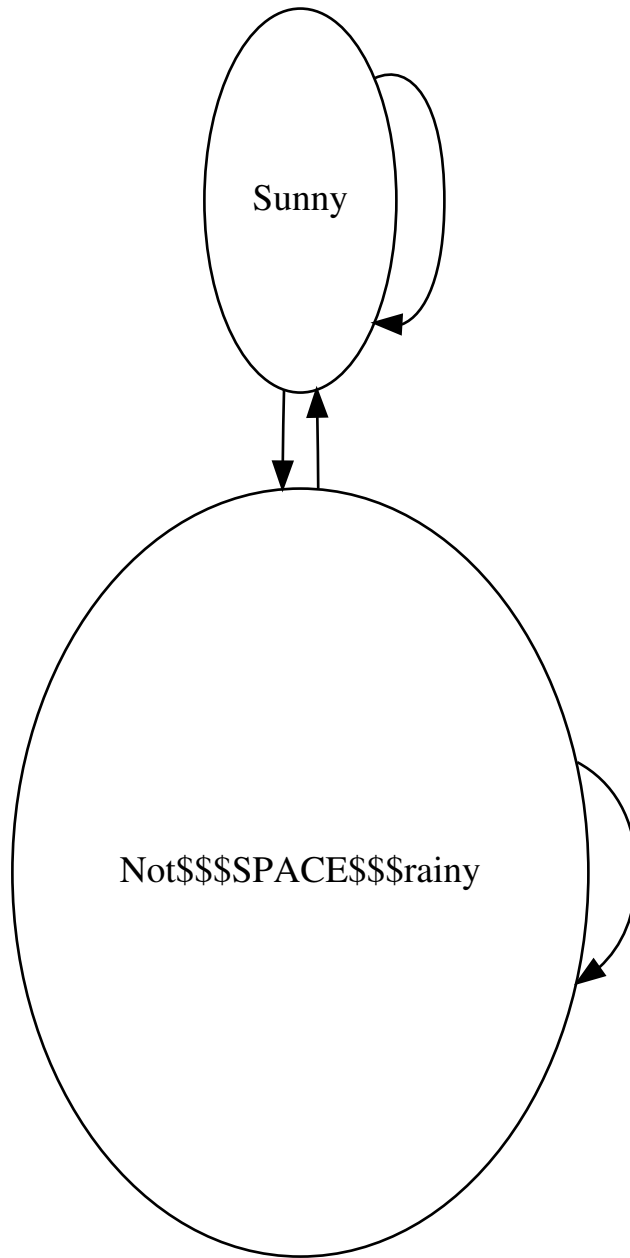


Figure 37: .svg file created from the ‘create_bundled_vertices_markov_chain’ function (algorithm 161) its .dot file, converted from .dot file to .svg using algorithm 339

8.7 Creating K_2 with bundled vertices

8.7.1 Graph

We reproduce the K_2 with named vertices of chapter 4.6 , but with our bundled vertices instead, as show in figure 38:

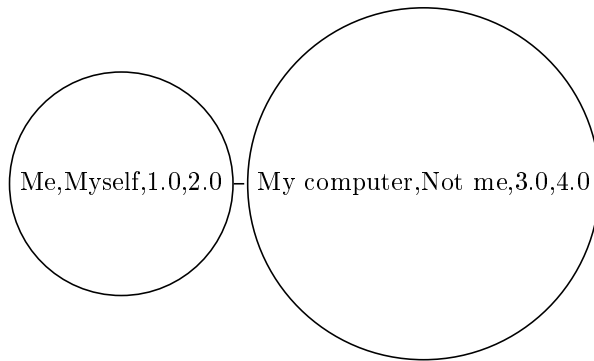


Figure 38: K_2 : a fully connected graph with two bundled vertices

8.7.2 Function to create such a graph

Algorithm 164 Creating K_2 as depicted in figure 18

```
#include "create_empty_undirected_bundled_vertices_graph.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    my_bundled_vertex
>
create_bundled_vertices_k2_graph() noexcept
{
    auto g = create_empty_undirected_bundled_vertices_graph
        ();
    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);
    const auto aer = boost::add_edge(vd_a, vd_b, g);
    assert(aer.second);
    g[vd_a] = my_bundled_vertex(
        "Me", "Myself", 1.0, 2.0
    );
    g[vd_b] = my_bundled_vertex(
        "My_computer", "Not_me", 3.0, 4.0
    );
    return g;
}
```

Most of the code is a slight modification of the ‘create_named_vertices_k2_graph’ function (algorithm 65). In the end, (references to) the my_bundled_vertices are obtained and set with two bundled my_bundled_vertex objects.

8.7.3 Creating such a graph

Demo:

Algorithm 165 Demo of the ‘create_bundled_vertices_k2_graph’ function (algorithm 164)

```
#include <cassert>
#include "create_bundled_vertices_k2_graph.h"
#include "has_bundled_vertex_with_my_vertex.h"

void create_bundled_vertices_k2_graph_demo() noexcept
{
    const auto g = create_bundled_vertices_k2_graph();
    assert(boost::num_edges(g) == 1);
    assert(boost::num_vertices(g) == 2);
    assert(has_bundled_vertex_with_my_vertex(
        my_bundled_vertex("Me", "Myself", 1.0, 2.0), g)
    );
    assert(has_bundled_vertex_with_my_vertex(
        my_bundled_vertex("My_computer", "Not_me", 3.0, 4.0), g)
    );
}
```

8.7.4 The .dot file produced

Algorithm 166 .dot file created from the ‘create_bundled_vertices_k2_graph’ function (algorithm 164), converted from graph to .dot file using algorithm 48

8.7.5 The .svg file produced

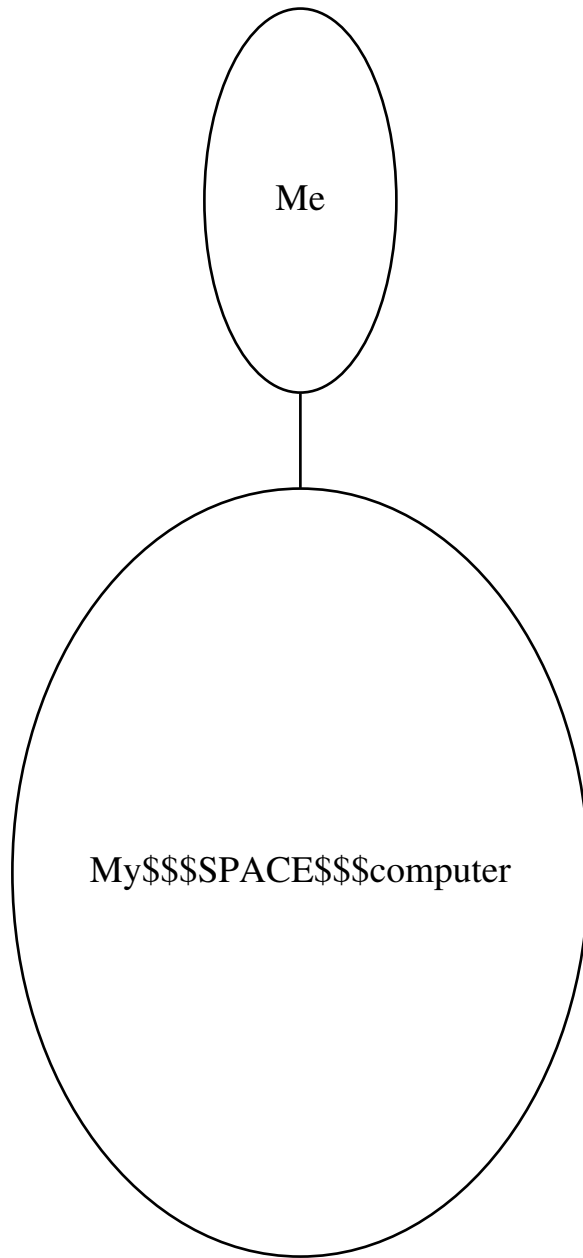


Figure 39: .svg file created from the ‘create_bundled_vertices_k2_graph’ function (algorithm 164) its .dot file, converted from .dot file to .svg using algorithm 339

9 Working on graphs with bundled vertices

When using graphs with bundled vertices, their state gives a way to find a vertex and working with it. This chapter shows some basic operations on graphs with bundled vertices.

- Check if there exists a vertex with a certain ‘my_bundled_vertex’: chapter 9.1
- Find a vertex with a certain ‘my_bundled_vertex’: chapter 9.2
- Get a vertex its ‘my_bundled_vertex’ from its vertex descriptor: chapter 9.3
- Set a vertex its ‘my_bundled_vertex’ using its vertex descriptor: chapter 9.4
- Setting all vertices their ‘my_bundled_vertex’-es: chapter 9.5
- Storing an directed/undirected graph with bundled vertices as a .dot file: chapter 9.6
- Loading a directed graph with bundled vertices from a .dot file: chapter 9.7
- Loading an undirected directed graph with bundled vertices from a .dot file: chapter 9.8

9.1 Has a bundled vertex with a my_bundled_vertex

Before modifying our vertices, let’s first determine if we can find a vertex by its bundled type (‘my_bundled_vertex’) in a graph. After obtain the vertex iterators, we can dereference each these to obtain the vertex descriptors and then compare each vertex its ‘my_bundled_vertex’ with the one desired.

Algorithm 167 Find if there is vertex with a certain `my_bundled_vertex`

```
#include <string>
#include <boost/graph/properties.hpp>
#include "my_bundled_vertex.h"

template <typename graph>
bool has_bundled_vertex_with_my_vertex(
    const my_bundled_vertex& v,
    const graph& g
) noexcept
{
    using vd = typename graph::vertex_descriptor;

    const auto vip = vertices(g);
    return std::find_if(vip.first, vip.second,
        [v, g](const vd& d)
        {
            return g[d] == v;
        }
    ) != vip.second;
}
```

This function can be demonstrated as in algorithm 168, where a certain `my_bundled_vertex` cannot be found in an empty graph. After adding the desired `my_bundled_vertex`, it is found.

Algorithm 168 Demonstration of the ‘has_bundled_vertex_with_my_vertex’ function

```
#include <cassert>
#include <iostream>

#include "add_bundled_vertex.h"
#include "create_empty_undirected_bundled_vertices_graph.h"
#include "has_bundled_vertex_with_my_vertex.h"
#include "my_bundled_vertex.h"

void has_bundled_vertex_with_my_vertex_demo() noexcept
{
    auto g = create_empty_undirected_bundled_vertices_graph();
    assert(!has_bundled_vertex_with_my_vertex(my_bundled_vertex("Felix"), g));
    add_bundled_vertex(my_bundled_vertex("Felix"), g);
    assert(has_bundled_vertex_with_my_vertex(my_bundled_vertex("Felix"), g));
}
```

Note that this function only finds if there is at least one bundled vertex with that my_bundled_vertex: it does not tell how many bundled vertices with that my_bundled_vertex exist in the graph.

9.2 Find a bundled vertex with a certain my_bundled_vertex

Where STL functions work with iterators, here we obtain a vertex descriptor (see chapter 2.6) to obtain a handle to the desired vertex. Algorithm 169 shows how to obtain a vertex descriptor to the first vertex found with a specific ‘my_bundled_vertex’ value.

Algorithm 169 Find the first vertex with a certain `my_bundled_vertex`

```
#include <cassert>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "has_bundled_vertex_with_my_vertex.h"
#include "my_bundled_vertex.h"

template <typename graph>
typename boost::graph_traits<graph>::vertex_descriptor
find_first_bundled_vertex_with_my_vertex(
    const my_bundled_vertex& v,
    const graph& g
) noexcept
{
    using vd = typename graph::vertex_descriptor;
    const auto vip = vertices(g);
    const auto i = std::find_if(
        vip.first, vip.second,
        [v,g](const vd d) { return g[d] == v; }
    );
    assert(i != vip.second);
    return *i;
}
```

With the vertex descriptor obtained, one can read and modify the vertex and the edges surrounding it. Algorithm 170 shows some examples of how to do so.

Algorithm 170 Demonstration of the ‘find_first_bundled_vertex_with_my_vertex’ function

```
#include <cassert>

#include "create_bundled_vertices_k2_graph.h"
#include "find_first_bundled_vertex_with_my_vertex.h"

void find_first_bundled_vertex_with_my_vertex_demo()
    noexcept
{
    const auto g = create_bundled_vertices_k2_graph();
    const auto vd =
        find_first_bundled_vertex_with_my_vertex(
            my_bundled_vertex("Me", "Myself", 1.0, 2.0),
            g
        );
    assert(out_degree(vd, g) == 1);
    assert(in_degree(vd, g) == 1);
}
```

9.3 Get a bundled vertex its ‘my_bundled_vertex’

To obtain the ‘my_bundled_vertex’ from a vertex descriptor is simple:

Algorithm 171 Get a bundled vertex its my_vertex from its vertex descriptor

```
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "my_bundled_vertex.h"

template <typename graph>
my_bundled_vertex get_my_bundled_vertex(
    const typename boost::graph_traits<graph>::
        vertex_descriptor& vd,
    const graph& g
) noexcept
{
    return g[vd];
}
```

One can just use the graph as a property map and let it be looked-up.

To use ‘get_bundled_vertex_my_vertex’, one first needs to obtain a vertex descriptor. Algorithm 172 shows a simple example.

Algorithm 172 Demonstration if the ‘get_bundled_vertex_my_vertex’ function

```
#include <cassert>
#include "add_bundled_vertex.h"
#include "create_empty_undirected_bundled_vertices_graph.h"
#include "find_first_bundled_vertex_with_my_vertex.h"
#include "get_my_bundled_vertex.h"

void get_my_bundled_vertex_demo() noexcept
{
    auto g
        = create_empty_undirected_bundled_vertices_graph();
    const my_bundled_vertex v{"Dex"};
    add_bundled_vertex(v, g);
    const auto vd
        = find_first_bundled_vertex_with_my_vertex(v, g);
    assert(get_my_bundled_vertex(vd, g) == v);
}
```

9.4 Set a bundled vertex its my_vertex

If you know how to get the ‘my_bundled_vertex’ from a vertex descriptor, setting it is just as easy, as shown in algorithm 173.

Algorithm 173 Set a bundled vertex its my_vertex from its vertex descriptor

```
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "my_bundled_vertex.h"

template <typename graph>
void set_my_bundled_vertex(
    const my_bundled_vertex& v,
    const typename boost::graph_traits<graph>::
        vertex_descriptor& vd,
    graph& g
) noexcept
{
    static_assert(!std::is_const<graph>::value, "graph_
        cannot_be_const");
    g[vd] = v;
}
```

To use ‘set_bundled_vertex_my_vertex’, one first needs to obtain a vertex descriptor. Algorithm 174 shows a simple example.

Algorithm 174 Demonstration if the ‘set_bundled_vertex_my_vertex’ function

```
#include <cassert>

#include "add_bundled_vertex.h"
#include "create_empty_undirected_bundled_vertices_graph.h"
#include "find_first_bundled_vertex_with_my_vertex.h"
#include "get_my_bundled_vertex.h"
#include "set_my_bundled_vertex.h"

void set_my_bundled_vertex_demo() noexcept
{
    auto g = create_empty_undirected_bundled_vertices_graph();
    const my_bundled_vertex old_name{"Dex"};
    add_bundled_vertex(old_name, g);
    const auto vd =
        find_first_bundled_vertex_with_my_vertex(old_name, g);
    assert(get_my_bundled_vertex(vd, g) == old_name);
    const my_bundled_vertex new_name{"Diggy"};
    set_my_bundled_vertex(new_name, vd, g);
    assert(get_my_bundled_vertex(vd, g) == new_name);
}
```

9.5 Setting all bundled vertices’ my_vertex objects

When the vertices of a graph are ‘my_bundled_vertex’ objects, one can set these as such:

Algorithm 175 Setting the bundled vertices' 'my_bundled_vertex'-es

```
#include <string>
#include <vector>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "my_bundled_vertex.h"

template <typename graph>
void set_my_bundled_vertexes(
    graph& g,
    const std::vector<my_bundled_vertex>& my_vertexes
) noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const");
    ;

    auto my_vertexes_begin = std::begin(my_vertexes);
    const auto my_vertexes_end = std::end(my_vertexes);
    const auto vip = vertices(g);
    const auto j = vip.second;
    for (
        auto i = vip.first;
        i!=j; ++i,
        ++my_vertexes_begin
    ) {
        assert(my_vertexes_begin != my_vertexes_end);
        g[*i] = *my_vertexes_begin;
    }
}
```

9.6 Storing a graph with bundled vertices as a .dot

If you used the 'create_bundled_vertices_k2_graph' function (algorithm 164) to produce a K_2 graph with vertices associated with 'my_bundled_vertex' objects, you can store these with algorithm 176:

Algorithm 176 Storing a graph with bundled vertices as a .dot file

```
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "make_bundled_vertices_writer.h"

template <typename graph>
void save_bundled_vertices_graph_to_dot(
    const graph& g,
    const std::string& filename
)
{
    std::ofstream f(filename);
    boost::write_graphviz(f, g,
        make_bundled_vertices_writer(g)
    );
}
```

This code looks small, because we call the ‘make_bundled_vertices_writer’ function, which is shown in algorithm 177:

Algorithm 177 The ‘make_bundled_vertices_writer’ function

```
template <typename graph>
inline bundled_vertices_writer<graph>
make_bundled_vertices_writer(
    const graph& g
)
{
    return bundled_vertices_writer<
        graph
    >(g);
}
```

Also this function is forwarding the real work to the ‘bundled_vertices_writer’, shown in algorithm 178:

Algorithm 178 The ‘bundled_vertices_writer’ function

```
#include <ostream>
#include "graphviz_encode.h"
#include "is_graphviz_friendly.h"

template <
    typename graph
>
class bundled_vertices_writer {
public:
    bundled_vertices_writer(
        graph g
    ) : m_g{g}
    {

    }

    template <class vertex_descriptor>
    void operator()(
        std::ostream& out,
        const vertex_descriptor& vd
    ) const noexcept {
        out
            << "[label=\"\"
            << graphviz_encode(
                m_g[vd].m_name
            )
            << "\",comment=\"\"
            << graphviz_encode(
                m_g[vd].m_description
            )
            << "\",width=\"
            << m_g[vd].m_x
            << "\",height=\"
            << m_g[vd].m_y
            << "]"
        ;
    }
private:
    graph m_g;
};
```

Here, some interesting things are happening: the writer needs the bundled property maps to work with and thus copies the whole graph to its internals. I have chosen to map the ‘my_bundled_vertex’ member variables to Graphviz

attributes (see chapter 25.2 for most Graphviz attributes) as shown in table 2:

| my_bundled_vertex variable | C++ data type | Graphviz data type | Graphviz attribute |
|----------------------------|---------------|--------------------|--------------------|
| m_name | std::string | string | label |
| m_description | std::string | string | comment |
| m_x | double | double | width |
| m_y | double | double | height |

Table 2: Mapping of my_bundled_vertex member variable and Graphviz attributes

Important in this mapping is that the C++ and the Graphviz data types match. I also chose attributes that matched as closely as possible.

The writer also encodes the std::string of the name and description to a Graphviz-friendly format. When loading the .dot file again, this will have to be undone again.

9.7 Loading a directed graph with bundled vertices from a .dot

When loading a graph from file, one needs to specify a type of graph. In this example, an directed graph with bundled vertices is loaded, as shown in algorithm 179:

Algorithm 179 Loading a directed graph with bundled vertices from a .dot file

```

#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "create_empty_directed_bundled_vertices_graph.h"
#include "graphviz_decode.h"
#include "is_regular_file.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    my_bundled_vertex
>
load_directed_bundled_vertices_graph_from_dot(
    const std::string& dot_filename
)
{
    assert(is_regular_file(dot_filename));
    std::ifstream f(dot_filename.c_str());
    auto g = create_empty_directed_bundled_vertices_graph();
        ;

    boost::dynamic_properties dp(boost::
        ignore_other_properties);
    dp.property("label", get(&my_bundled_vertex::m_name, g));
        ;
    dp.property("comment", get(&my_bundled_vertex::
        m_description, g));
    dp.property("width", get(&my_bundled_vertex::m_x, g));
    dp.property("height", get(&my_bundled_vertex::m_y, g));
    boost::read_graphviz(f, g, dp);

    //Decode vertices
    const auto vip = vertices(g);
    const auto j = vip.second;
    for (auto i = vip.first; i!=j; ++i)
    {
        g[*i].m_name = graphviz_decode(g[*i].m_name);
        g[*i].m_description = graphviz_decode(g[*i].
            m_description);
    }

    return g;
}

```

In this algorithm, first it is checked if the file to load exists. Then an empty directed graph is created, to save typing the typename explicitly.

Then a `boost::dynamic_properties` is created with its default constructor, after which we set it to follow the same mapping as in the previous chapter. From this and the empty graph, `'boost::read_graphviz'` is called to build up the graph.

At the moment the graph is created, all `'my_bundled_vertex'` their names and description are in a Graphviz-friendly format. By obtaining all vertex iterators and vertex descriptors, the encoding is made undone.

Algorithm 180 shows how to use the `'load_directed_bundled_vertices_graph_from_dot'` function:

Algorithm 180 Demonstration of the `'load_directed_bundled_vertices_graph_from_dot'` function

```
#include "create_bundled_vertices_markov_chain.h"
#include "load_directed_bundled_vertices_graph_from_dot.h"
"

#include "save_bundled_vertices_graph_to_dot.h"
#include "get_my_bundled_vertexes.h"

void load_directed_bundled_vertices_graph_from_dot_demo()
    noexcept
{
    using boost::num_edges;
    using boost::num_vertices;

    const auto g
        = create_bundled_vertices_markov_chain();
    const std::string filename{
        "create_bundled_vertices_markov_chain.dot"
    };
    save_bundled_vertices_graph_to_dot(g, filename);
    const auto h
        = load_directed_bundled_vertices_graph_from_dot(
            filename);
    assert(num_edges(g) == num_edges(h));
    assert(num_vertices(g) == num_vertices(h));
    assert(get_my_bundled_vertexes(g) ==
        get_my_bundled_vertexes(h));
}
```

This demonstration shows how the Markov chain is created using the `'create_bundled_vertices_markov_chain'` function (algorithm 161), saved and then loaded. The loaded graph is checked to be the same as the original.

9.8 Loading an undirected graph with bundled vertices from a .dot

When loading a graph from file, one needs to specify a type of graph. In this example, an undirected graph with bundled vertices is loaded, as shown in algorithm 181:

Algorithm 181 Loading an undirected graph with bundled vertices from a .dot file

```

#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "create_empty_undirected_bundled_vertices_graph.
    h"
#include "graphviz_decode.h"
#include "is_regular_file.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    my_bundled_vertex
>
load_undirected_bundled_vertices_graph_from_dot(
    const std::string& dot_filename
)
{
    assert(is_regular_file(dot_filename));
    std::ifstream f(dot_filename.c_str());
    auto g = create_empty_undirected_bundled_vertices_graph
        ();

    boost::dynamic_properties dp(boost::
        ignore_other_properties);
    dp.property("label", get(&my_bundled_vertex::m_name, g))
        ;
    dp.property("comment", get(&my_bundled_vertex::
        m_description, g));
    dp.property("width", get(&my_bundled_vertex::m_x, g));
    dp.property("height", get(&my_bundled_vertex::m_y, g));
    boost::read_graphviz(f, g, dp);

    //Decode vertices
    const auto vip = vertices(g);
    const auto j = vip.second;
    for (auto i = vip.first; i!=j; ++i)
    {
        g[*i].m_name = graphviz_decode(g[*i].m_name);
        g[*i].m_description = graphviz_decode(g[*i].
            m_description);
    }

    return g;
}

```

The only difference with loading a directed graph, is that the initial empty graph is undirected instead. Chapter 9.7 describes the rationale of this function.

Algorithm 182 shows how to use the ‘load_undirected_bundled_vertices_graph_from_dot’ function:

Algorithm 182 Demonstration of the ‘load_undirected_bundled_vertices_graph_from_dot’ function

```
#include <cassert>
#include "create_bundled_vertices_k2_graph.h"
#include "load_undirected_bundled_vertices_graph_from_dot
.h"
#include "save_bundled_vertices_graph_to_dot.h"
#include "get_my_bundled_vertexes.h"

void load_undirected_bundled_vertices_graph_from_dot_demo
() noexcept
{
    using boost::num_edges;
    using boost::num_vertices;

    const auto g
        = create_bundled_vertices_k2_graph();
    const std::string filename{
        "create_bundled_vertices_k2_graph.dot"
    };
    save_bundled_vertices_graph_to_dot(g, filename);
    const auto h
        = load_undirected_bundled_vertices_graph_from_dot(
            filename);
    assert(get_my_bundled_vertexes(g)
        == get_my_bundled_vertexes(h)
    );
}
```

This demonstration shows how K_2 with bundled vertices is created using the ‘create_bundled_vertices_k2_graph’ function (algorithm 164), saved and then loaded. The loaded graph is checked to be the same as the original.

10 Building graphs with bundled edges and vertices

Up until now, the graphs created have had only bundled vertices. In this chapter, graphs will be created, in which both the edges and vertices have a bundled

‘my_bundled_edge’ and ‘my_bundled_edge’ type⁹.

- An empty directed graph that allows for bundled edges and vertices: see chapter 10.2
- An empty undirected graph that allows for bundled edges and vertices: see chapter 10.3
- A two-state Markov chain with bundled edges and vertices: see chapter 10.6
- K_3 with bundled edges and vertices: see chapter 10.7

In the process, some basic (sometimes bordering trivial) functions are shown:

- Creating the ‘my_bundled_edge’ class: see chapter 10.1
- Adding a bundled ‘my_bundled_edge’: see chapter 10.4

These functions are mostly there for completion and showing which data types are used.

10.1 Creating the bundled edge class

In this example, I create a ‘my_bundled_edge’ class. Here I will show the header file of it, as the implementation of it is not important yet.

⁹I do not intend to be original in naming my data types

Algorithm 183 Declaration of `my_bundled_edge`

```
#include <string>
#include <iosfwd>

class my_bundled_edge
{
public:
    explicit my_bundled_edge(
        const std::string& name = "",
        const std::string& description = "",
        const double width = 1.0,
        const double height = 1.0
    ) noexcept;
    std::string m_name;
    std::string m_description;
    double m_width;
    double m_height;
};

bool operator==(const my_bundled_edge& lhs, const
    my_bundled_edge& rhs) noexcept;
bool operator!=(const my_bundled_edge& lhs, const
    my_bundled_edge& rhs) noexcept;
```

`my_bundled_edge` is a class that has multiple properties: two doubles ‘`m_width`’ (‘`m_`’ stands for member) and ‘`m_height`’, and two `std::string`s `m_name` and `m_description`. ‘`my_bundled_edge`’ is copyable, but cannot trivially be converted to a ‘`std::string`.’ ‘`my_bundled_edge`’ is comparable for equality (that is, `operator==` is defined).

‘`my_bundled_edge`’ does not have to have the stream operators defined for file I/O, as this goes via the public member variables.

10.2 Create an empty directed graph with bundled edges and vertices

Algorithm 184 Creating an empty directed graph with bundled edges and vertices

```
#include <boost/graph/adjacency_list.hpp>
#include "my_bundled_edge.h"
#include "my_bundled_vertex.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    my_bundled_vertex,
    my_bundled_edge
>
create_empty_directed_bundled_edges_and_vertices_graph()
    noexcept
{
    return {};
}
```

This code is very similar to the code described in chapter 12.3, except that there is a new, fifth template argument:

```
boost::property<boost::edge_bundled_type_t, my_edge>
```

This can be read as: “edges have the property ‘boost::edge_bundled_type_t’, which is of data type ‘my_bundled_edge’”. Or simply: “edges have a bundled type called my_bundled_edge”.

Demo:

Algorithm 185 Demonstration of the ‘create_empty_directed_bundled_edges_and_vertices_graph’ function

```
#include "
    create_empty_directed_bundled_edges_and_vertices_graph
    .h"

void
    create_empty_directed_bundled_edges_and_vertices_graph_demo
    () noexcept
{
    const auto g =
        create_empty_directed_bundled_edges_and_vertices_graph
        ();
    assert(boost::num_edges(g) == 0);
    assert(boost::num_vertices(g) == 0);
}
```

10.3 Create an empty undirected graph with bundled edges and vertices

Algorithm 186 Creating an empty undirected graph with bundled edges and vertices

```
#include <boost/graph/adjacency_list.hpp>
#include "my_bundled_edge.h"
#include "my_bundled_vertex.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    my_bundled_vertex,
    my_bundled_edge
>
create_empty_undirected_bundled_edges_and_vertices_graph
    () noexcept
{
    return {};
}
```

This code is very similar to the code described in chapter 10.2, except that the directedness (the third template argument) is undirected (due to the `boost::undirectedS`).
Demo:

Algorithm 187 Demonstration of the ‘create_empty_undirected_bundled_edges_and_vertices_graph’ function

```
#include <cassert>
#include "
    create_empty_undirected_bundled_edges_and_vertices_graph
    .h"

void
    create_empty_undirected_bundled_edges_and_vertices_graph_demo
    () noexcept
{
    const auto g
        =
            create_empty_undirected_bundled_edges_and_vertices_graph
            ();
    assert(boost::num_edges(g) == 0);
    assert(boost::num_vertices(g) == 0);
}
```

10.4 Add a bundled edge

Adding a bundled edge is very similar to adding a named edge (chapter 6.3).

Algorithm 188 Add a bundled edge

```
#include <cassert>
#include <boost/graph/adjacency_list.hpp>
#include "my_bundled_edge.h"

template <typename graph>
typename boost::graph_traits<graph>::edge_descriptor
add_bundled_edge(
    const typename boost::graph_traits<graph>::
        vertex_descriptor& vd_from,
    const typename boost::graph_traits<graph>::
        vertex_descriptor& vd_to,
    const my_bundled_edge& v,
    graph& g
) noexcept
{
    static_assert(!std::is_const<graph>::value, "graph_
        cannot_be_const");
    const auto aer = boost::add_edge(vd_from, vd_to, g);
    assert(aer.second);
    g[aer.first] = v;
    return aer.first;
}
```

When having added a new (abstract) edge to the graph, the edge descriptor is used to set the `my_edge` in the graph.

Here is the demo:

Algorithm 189 Demo of ‘add_bundled_edge’

```
#include <cassert>
#include "add_bundled_edge.h"
#include "add_bundled_vertex.h"
#include "
    create_empty_directed_bundled_edges_and_vertices_graph
    .h"
// #include "my_bundled_vertex.h"

void add_bundled_edge_demo() noexcept
{
    auto g =
        create_empty_directed_bundled_edges_and_vertices_graph
        ();
    const auto vd_from = add_bundled_vertex(
        my_bundled_vertex("From"), g);
    const auto vd_to = add_bundled_vertex(my_bundled_vertex
        ("To"), g);
    add_bundled_edge(vd_from, vd_to, my_bundled_edge("X"),
        g);
    assert(boost::num_vertices(g) == 2);
    assert(boost::num_edges(g) == 1);
}
```

10.5 Getting the bundled edges my_edges

When the edges of a graph are ‘my_bundled_edge’ objects, one can extract these all as such:

Algorithm 190 Get the edges' my_bundled_edges

```
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include "my_bundled_edge.h"

template <typename graph>
std::vector<my_bundled_edge> get_my_bundled_edges(
    const graph& g
) noexcept
{
    using ed = typename boost::graph_traits<graph>::
        edge_descriptor;
    std::vector<my_bundled_edge> v(boost::num_edges(g));
    const auto eip = edges(g);
    std::transform(eip.first, eip.second, std::begin(v),
        [g](const ed e) { return g[e]; }
    );
    return v;
}
```

The 'my_bundled_edge' object associated with the edges are obtained from the graph its property_map and then put into a std::vector.

Note: the order of the my_bundled_edge objects may be different after saving and loading.

When trying to get the edges' my_bundled_edge objects from a graph without bundled edges objects associated, you will get the error 'formed reference to void' (see chapter 24.1).

10.6 Creating a Markov-chain with bundled edges and vertices

10.6.1 Graph

Figure 40 shows the graph that will be reproduced:

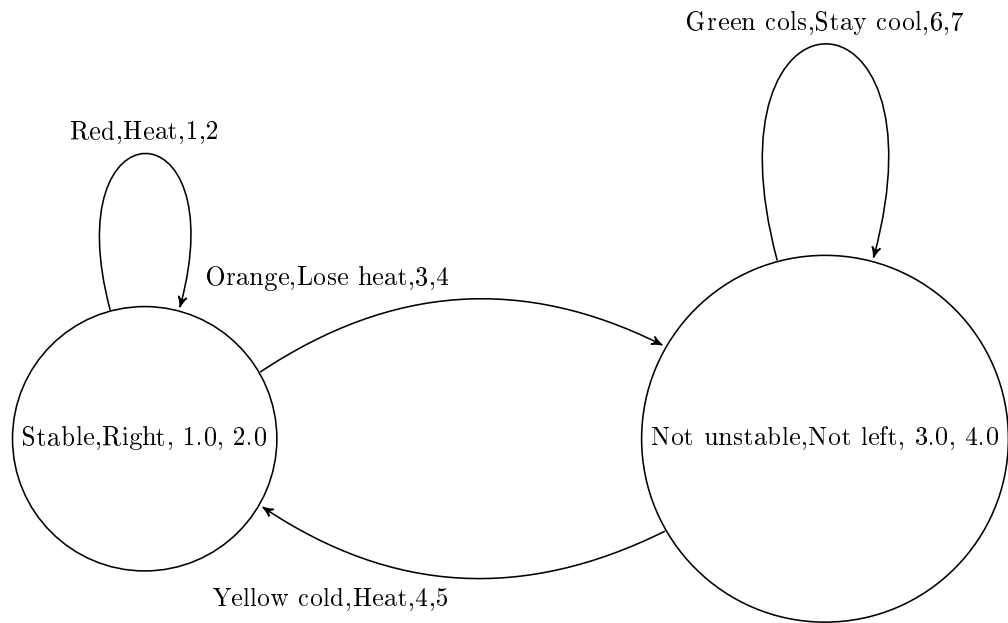


Figure 40: A two-state Markov chain where the edges and vertices have bundled properties. The edges' and vertices' properties are nonsensical

10.6.2 Function to create such a graph

Here is the code creating a two-state Markov chain with bundled edges and vertices:

Algorithm 191 Creating the two-state Markov chain as depicted in figure 40

```

#include <cassert>
#include "
    create_empty_directed_bundled_edges_and_vertices_graph
    .h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    my_bundled_vertex,
    my_bundled_edge
>
create_bundled_edges_and_vertices_markov_chain() noexcept
{
    auto g
        =
            create_empty_directed_bundled_edges_and_vertices_graph
            ();
    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);
    const auto aer_aa = boost::add_edge(vd_a, vd_a, g);
    assert(aer_aa.second);
    const auto aer_ab = boost::add_edge(vd_a, vd_b, g);
    assert(aer_ab.second);
    const auto aer_ba = boost::add_edge(vd_b, vd_a, g);
    assert(aer_ba.second);
    const auto aer_bb = boost::add_edge(vd_b, vd_b, g);
    assert(aer_bb.second);

    g[vd_a]
        = my_bundled_vertex("Stable", "Right", 1.0, 2.0);
    g[vd_b]
        = my_bundled_vertex("Not_unstable", "Not_left",
            , 3.0, 4.0);

    g[aer_aa.first]
        = my_bundled_edge("Red", "Heat", 1.0, 2.0);
    g[aer_ab.first]
        = my_bundled_edge("Orange", "Lose_heat", 3.0, 4.0);
    g[aer_ba.first]
        = my_bundled_edge("Yellow_cold", "Heat", 5.0, 6.0);
    g[aer_bb.first]
        = my_bundled_edge("Green_cold", "Stay_cool", 7.0, 8.0);

    return g;
}

```


10.6.3 Creating such a graph

Here is the demo:

Algorithm 192 Demo of the ‘create_bundled_edges_and_vertices_markov_chain’ function (algorithm 191)

```
#include <cassert>
#include "create_bundled_edges_and_vertices_markov_chain.h"
#include "get_my_bundled_edges.h"
#include "my_bundled_vertex.h"

void create_bundled_edges_and_vertices_markov_chain_demo
() noexcept
{
    const auto g =
        create_bundled_edges_and_vertices_markov_chain();
    const std::vector<my_bundled_edge> edge_my_edges{
        get_my_bundled_edges(g)
    };
    const std::vector<my_bundled_edge> expected_my_edges{
        my_bundled_edge("Red","Heat",1.0,2.0),
        my_bundled_edge("Orange","Lose_heat",3.0,4.0),
        my_bundled_edge("Yellow_cold","Heat",5.0,6.0),
        my_bundled_edge("Green_cold","Stay_cool",7.0,8.0)
    };
    assert(edge_my_edges == expected_my_edges);
}
```

10.6.4 The .dot file produced

Algorithm 193 .dot file created from the ‘create_bundled_edges_and_vertices_markov_chain’ function (algorithm 191), converted from graph to .dot file using algorithm 48

```
digraph G {
0[label="Stable",comment="Right",width=1,height=2];
1[label="Not$$$SPACE$$$unstable",comment="Not$$$SPACE$$$left",width=3,height=4];
0->0 [label="Red",comment="Heat",width=1,height=2];
0->1 [label="Orange",comment="Lose$$$SPACE$$$heat",width=3,height=4];
1->0 [label="Yellow$$$SPACE$$$cold",comment="Heat",width=5,height=6];
1->1 [label="Green$$$SPACE$$$cold",comment="Stay$$$SPACE$$$cool",width=7,height=8];
}
```

10.6.5 The .svg file produced

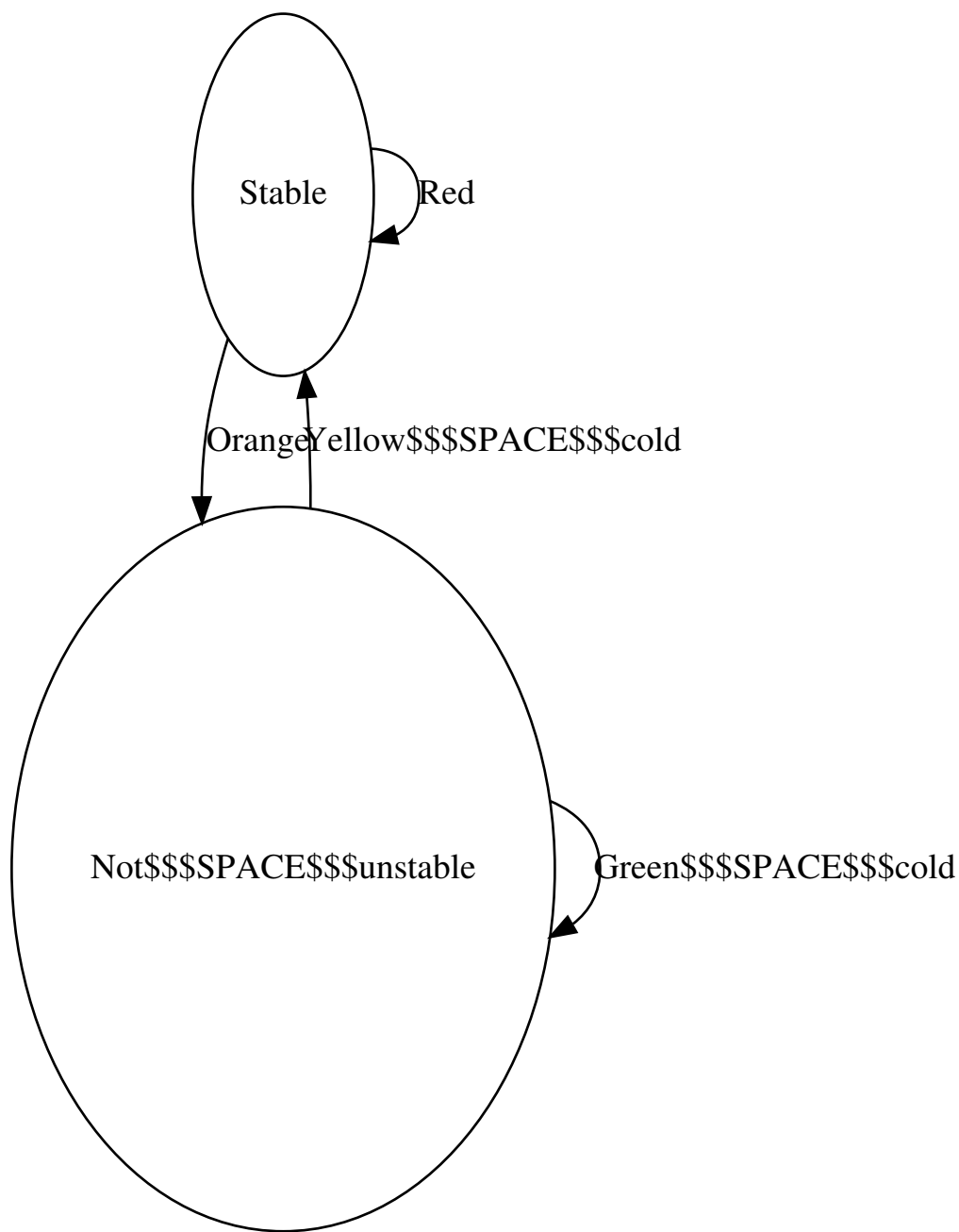


Figure 41: .svg file created from the 'create_bundled_edges_and_vertices_markov_chain' function (algorithm 220) its .dot file, converted from .dot file to .svg using algorithm 339

10.7 Creating K_3 with bundled edges and vertices

Instead of using edges with a name, or other properties, here we use a bundled edge class called 'my_bundled_edge'.

10.7.1 Graph

We reproduce the K_3 with named edges and vertices of chapter 6.8 , but with our bundled edges and vertices instead:

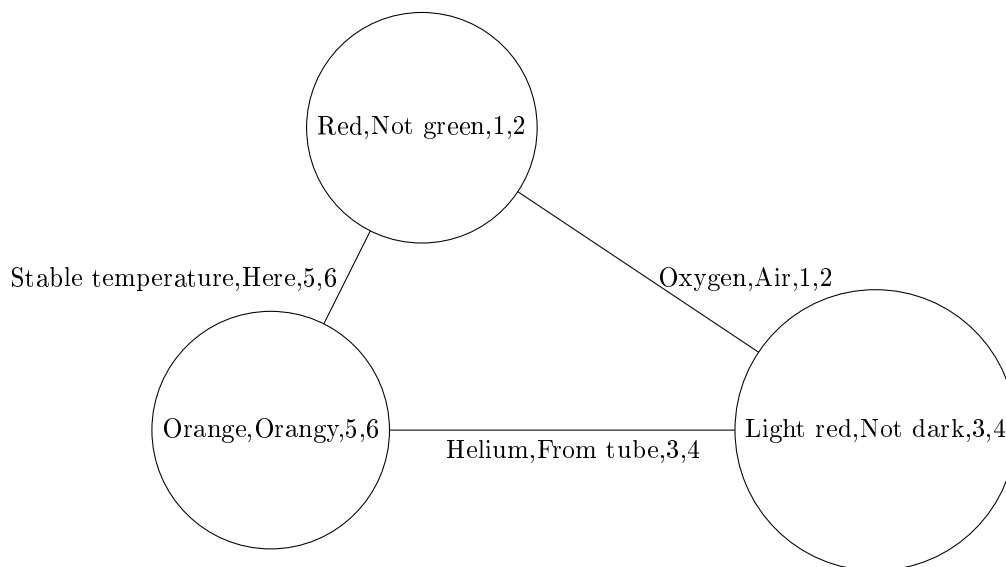


Figure 42: K_3 : a fully connected graph with three named edges and vertices

10.7.2 Function to create such a graph

Algorithm 194 Creating K_3 as depicted in figure 30

```
#include "
    create_empty_undirected_bundled_edges_and_vertices_graph
    .h"
#include "add_bundled_edge.h"
#include "add_bundled_vertex.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    my_bundled_vertex,
    my_bundled_edge
>
create_bundled_edges_and_vertices_k3_graph() noexcept
{
    auto g
        =
            create_empty_undirected_bundled_edges_and_vertices_graph
            ();
    const auto vd_a = add_bundled_vertex(
        my_bundled_vertex("Red", "Not_green", 1.0, 2.0),
        g
    );
    const auto vd_b = add_bundled_vertex(
        my_bundled_vertex("Light_red", "Not_dark", 3.0, 4.0),
        g
    );
    const auto vd_c = add_bundled_vertex(
        my_bundled_vertex("Orange", "Orangy", 5.0, 6.0),
        g
    );
    add_bundled_edge(vd_a, vd_b,
        my_bundled_edge("Oxygen", "Air", 1.0, 2.0),
        g
    );
    add_bundled_edge(vd_b, vd_c,
        my_bundled_edge("Helium", "From_tube", 3.0, 4.0),
        g
    );
    add_bundled_edge(vd_c, vd_a,
        my_bundled_edge("Stable_temperature", "Here", 5.0, 6.0),
        g
    );
    return g;
}
```

Most of the code is a slight modification of algorithm 128. In the end, the `my_edges` and `my_vertices` are obtained as the graph its `property_map` and set with the `'my_bundled_edge'` and `'my_bundled_vertex'` objects.

10.7.3 Creating such a graph

Here is the demo:

Algorithm 195 Demo of the `'create_bundled_edges_and_vertices_k3_graph'` function (algorithm 194)

```
#include <cassert>
#include "create_bundled_edges_and_vertices_k3_graph.h"

void create_bundled_edges_and_vertices_k3_graph_demo()
    noexcept
{
    auto g
        = create_bundled_edges_and_vertices_k3_graph();
    assert(boost::num_edges(g) == 3);
    assert(boost::num_vertices(g) == 3);
}
```

10.7.4 The .dot file produced

Algorithm 196 .dot file created from the `'create_bundled_edges_and_vertices_markov_chain'` function (algorithm 194), converted from graph to .dot file using algorithm 48

```
graph G {
0[label="Red",comment="Not$$$$SPACE$$$green",width=1,height=2];
1[label="Light$$$$SPACE$$$red",comment="Not$$$$SPACE$$$dark",width=3,height=4];
2[label="Orange",comment="Orangy",width=5,height=6];
0--1 [label="Oxygen",comment="Air",width=1,height=2];
1--2 [label="Helium",comment="From$$$$SPACE$$$tube",width=3,height=4];
2--0 [label="Stable$$$$SPACE$$$temperature",comment="Here",width=5,height=6];
}
```

10.7.5 The .svg file produced

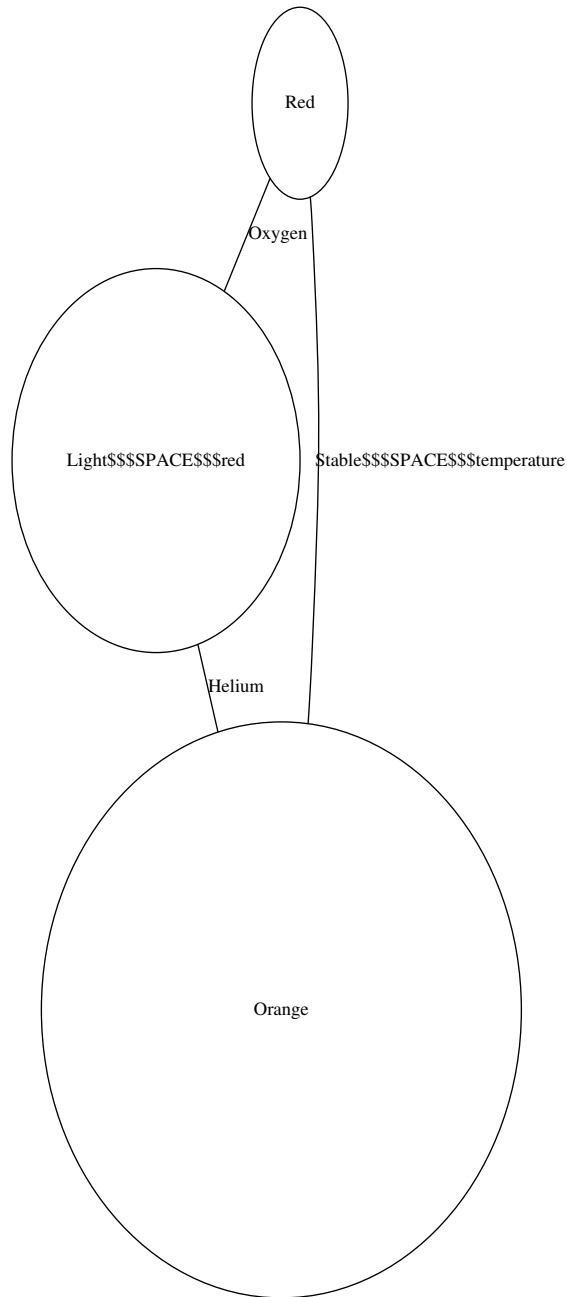


Figure 43: .svg file created from the 'create_bundled_edges_and_vertices_k3_graph' function (algorithm 220) its .dot file, converted from .dot file to .svg using algorithm 339

11 Working on graphs with bundled edges and vertices

11.1 Has a my_bundled_edge

Before modifying our edges, let's first determine if we can find an edge by its bundled type ('my_bundled_edge') in a graph. After obtaining a my_bundled_edge map, we obtain the edge iterators, dereference these to obtain the edge descriptors and then compare each edge its my_bundled_edge with the one desired.

Algorithm 197 Find if there is a bundled edge with a certain my_bundled_edge

```
#include <boost/graph/properties.hpp>
#include "my_bundled_edge.h"

template <typename graph>
bool has_bundled_edge_with_my_edge(
    const my_bundled_edge& e,
    const graph& g
) noexcept
{
    using ed = typename boost::graph_traits<graph>::
        edge_descriptor;
    const auto eip = edges(g);
    return std::find_if(eip.first, eip.second,
        [e, g](const ed& d)
        {
            return g[d] == e;
        }) != eip.second;
}
```

This function can be demonstrated as in algorithm 198, where a certain 'my_bundled_edge' cannot be found in an empty graph. After adding the desired my_bundled_edge, it is found.

Algorithm 198 Demonstration of the ‘has_bundled_edge_with_my_edge’ function

```
#include <cassert>
#include "create_bundled_edges_and_vertices_k3_graph.h"
#include "has_bundled_edge_with_my_edge.h"

void has_bundled_edge_with_my_edge_demo() noexcept
{
    auto g
        = create_bundled_edges_and_vertices_k3_graph();
    assert(
        has_bundled_edge_with_my_edge(
            my_bundled_edge("Oxygen", "Air", 1.0, 2.0), g
        )
    );
}
```

Note that this function only finds if there is at least one edge with that my_bundled_edge: it does not tell how many edges with that my_bundled_edge exist in the graph.

11.2 Find a my_bundled_edge

Where STL functions work with iterators, here we obtain an edge descriptor (see chapter 2.12) to obtain a handle to the desired edge. Algorithm 199 shows how to obtain an edge descriptor to the first edge found with a specific my_bundled_edge value.

Algorithm 199 Find the first bundled edge with a certain `my_bundled_edge`

```
#include <cassert>
#include <boost/graph/graph_traits.hpp>
#include "has_bundled_edge_with_my_edge.h"
#include "has_custom_edge_with_my_edge.h"
#include "my_bundled_edge.h"

template <typename graph>
typename boost::graph_traits<graph>::edge_descriptor
find_first_bundled_edge_with_my_edge(
    const my_bundled_edge& e,
    const graph& g
) noexcept
{
    using ed = typename boost::graph_traits<graph>::
        edge_descriptor;
    const auto eip = edges(g);
    const auto i = std::find_if(
        eip.first, eip.second,
        [e,g](const ed d) { return g[d] == e; }
    );
    assert(i != eip.second);
    return *i;
}
```

With the edge descriptor obtained, one can read and modify the edge and the vertices surrounding it. Algorithm 200 shows some examples of how to do so.

Algorithm 200 Demonstration of the ‘find_first_bundled_edge_with_my_edge’ function

```
#include <cassert>

#include "create_bundled_edges_and_vertices_k3_graph.h"
#include "find_first_bundled_edge_with_my_edge.h"

void find_first_bundled_edge_with_my_edge_demo() noexcept
{
    const auto g
        = create_bundled_edges_and_vertices_k3_graph();
    const auto ed
        = find_first_bundled_edge_with_my_edge(
            my_bundled_edge("Oxygen", "Air", 1.0, 2.0),
            g
        );
    assert(boost::source(ed, g)
        != boost::target(ed, g)
    );
}
```

11.3 Get an edge its my_bundled_edge

To obtain the my_bundled_edge from an edge descriptor, one needs to pull out the my_bundled_edges map and then look up the my_edge of interest.

Algorithm 201 Get a vertex its my_bundled_vertex from its vertex descriptor

```
#include <boost/graph/graph_traits.hpp>
#include "my_bundled_edge.h"

template <typename graph>
my_bundled_edge get_my_bundled_edge(
    const typename boost::graph_traits<graph>::
        edge_descriptor& ed,
    const graph& g
) noexcept
{
    return g[ed];
}
```

To use ‘get_my_bundled_edge’, one first needs to obtain an edge descriptor. Algorithm 202 shows a simple example.

Algorithm 202 Demonstration if the ‘get_my_bundled_edge’ function

```
#include <cassert>

#include "add_bundled_edge.h"
#include "add_bundled_vertex.h"
#include "
    create_empty_undirected_bundled_edges_and_vertices_graph
    .h"
#include "find_first_bundled_edge_with_my_edge.h"
#include "get_my_bundled_edge.h"

void get_my_bundled_edge_demo() noexcept
{
    auto g
        =
            create_empty_undirected_bundled_edges_and_vertices_graph
            ();
    const my_bundled_edge edge{"Dex"};
    const auto vd_a = add_bundled_vertex(
        my_bundled_vertex("A"), g
    );
    const auto vd_b = add_bundled_vertex(
        my_bundled_vertex("B"), g
    );
    add_bundled_edge(vd_a, vd_b, edge, g);
    const auto ed
        = find_first_bundled_edge_with_my_edge(edge, g);
    assert(get_my_bundled_edge(ed, g) == edge);
}
```

11.4 Set an edge its my_bundled_edge

If you know how to get the my_bundled_edge from an edge descriptor, setting it is just as easy, as shown in algorithm 203.

Algorithm 203 Set a bundled edge its `my_bundled_edge` from its edge descriptor

```
#include <boost/graph/properties.hpp>
#include "my_bundled_edge.h"

template <typename graph>
void set_my_bundled_edge(
    const my_bundled_edge& name,
    const typename boost::graph_traits<graph>::
        edge_descriptor& ed,
    graph& g
) noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const");
    g[ed] = name;
}
```

To use ‘`set_bundled_edge_my_edge`’, one first needs to obtain an edge descriptor. Algorithm 204 shows a simple example.

Algorithm 204 Demonstration if the ‘set_bundled_edge_my_edge’ function

```
#include <cassert>

#include "add_bundled_edge.h"
#include "add_bundled_vertex.h"
#include "
    create_empty_undirected_bundled_edges_and_vertices_graph
    .h"
#include "find_first_bundled_edge_with_my_edge.h"
#include "get_my_bundled_edge.h"
#include "set_my_bundled_edge.h"

void set_my_bundled_edge_demo() noexcept
{
    auto g
        =
            create_empty_undirected_bundled_edges_and_vertices_graph
            ();
    const auto vd_a = add_bundled_vertex(my_bundled_vertex{
        "A"}, g);
    const auto vd_b = add_bundled_vertex(my_bundled_vertex{
        "B"}, g);
    const my_bundled_edge old_edge{"Dex"};
    add_bundled_edge(vd_a, vd_b, old_edge, g);
    const auto vd
        = find_first_bundled_edge_with_my_edge(old_edge, g);
    assert(get_my_bundled_edge(vd, g)
        == old_edge
    );
    const my_bundled_edge new_edge{"Diggy"};
    set_my_bundled_edge(new_edge, vd, g);
    assert(get_my_bundled_edge(vd, g)
        == new_edge
    );
}
```

11.5 Storing a graph with bundled edges and vertices as a .dot

If you used the ‘create_bundled_edges_and_vertices_k3_graph’ function (algorithm 194) to produce a K_3 graph with edges and vertices associated with my_bundled_edge and my_bundled_vertex objects, you can store these my_bundled_edges and my_bundled_vertex-es additionally with algorithm 205:

Algorithm 205 Storing a graph with bundled edges and vertices as a .dot file

```
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "make_bundled_vertices_writer.h"
#include "make_bundled_edges_writer.h"

template <typename graph>
void save_bundled_edges_and_vertices_graph_to_dot(
    const graph& g,
    const std::string& filename
)
{
    std::ofstream f(filename);
    boost::write_graphviz(
        f,
        g,
        make_bundled_vertices_writer(g),
        make_bundled_edges_writer(g)
    );
}
```

11.6 Load a directed graph with bundled edges and vertices from a .dot file

When loading a graph from file, one needs to specify a type of graph. In this example, an directed graph with bundled edges and vertices is loaded, as shown in algorithm 206:

Algorithm 206 Loading a directed graph with bundled edges and vertices from a .dot file

```

#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "
    create_empty_directed_bundled_edges_and_vertices_graph
    .h"
#include "is_regular_file.h"
#include "graphviz_decode.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    my_bundled_vertex,
    my_bundled_edge
>
load_directed_bundled_edges_and_vertices_graph_from_dot(
    const std::string& dot_filename
)
{
    assert(is_regular_file(dot_filename));
    std::ifstream f(dot_filename.c_str());
    auto g =
        create_empty_directed_bundled_edges_and_vertices_graph
        ();

    boost::dynamic_properties dp(boost::
        ignore_other_properties);
    dp.property("label", get(&my_bundled_vertex::m_name, g))
        ;
    dp.property("comment", get(&my_bundled_vertex::
        m_description, g));
    dp.property("width", get(&my_bundled_vertex::m_x, g));
    dp.property("height", get(&my_bundled_vertex::m_y, g));
    dp.property("edge_id", get(&my_bundled_edge::m_name, g))
        ;
    dp.property("label", get(&my_bundled_edge::m_name, g));
    dp.property("comment", get(&my_bundled_edge::
        m_description, g));
    dp.property("width", get(&my_bundled_edge::m_width, g))
        ;
    dp.property("height", get(&my_bundled_edge::m_height, g
        ));
    boost::read_graphviz(f, g, dp);

    //Decode vertices
    {
        226
        const auto vip = vertices(g);
        const auto j = vip.second;
        for (auto i = vip.first; i!=j; ++i)
        {
            g[*i].m_name = graphviz_decode(g[*i].m_name);
            g[*i].m_description = graphviz_decode(g[*i].
                m_description);
        }
    }
}

```

In this algorithm, first it is checked if the file to load exists. Then an empty directed graph is created. Next to this, a `boost::dynamic_properties` is created with its default constructor, after which we direct the `boost::dynamic_properties` to find a `'node_id'` and `'label'` in the vertex name map, `'edge_id'` and `'label'` to the edge name map. From this and the empty graph, `'boost::read_graphviz'` is called to build up the graph.

Algorithm 207 shows how to use the `'load_directed_bundled_edges_and_vertices_graph_from_dot'` function:

Algorithm 207 Demonstration of the ‘load_directed_bundled_edges_and_vertices_graph_from_dot’ function

```
#include "create_bundled_edges_and_vertices_markov_chain.h"
#include "get_sorted_bundled_vertex_my_vertexes.h"
#include "load_directed_bundled_edges_and_vertices_graph_from_dot.h"
#include "save_bundled_edges_and_vertices_graph_to_dot.h"

void
load_directed_bundled_edges_and_vertices_graph_from_dot_demo
() noexcept
{
    using boost::num_edges;
    using boost::num_vertices;

    const auto g
        = create_bundled_edges_and_vertices_markov_chain();
    const std::string filename{
        "create_bundled_edges_and_vertices_markov_chain.dot"
    };
    save_bundled_edges_and_vertices_graph_to_dot(g,
        filename);
    const auto h
        =
            load_directed_bundled_edges_and_vertices_graph_from_dot
            (
                filename
            );
    assert(num_edges(g) == num_edges(h));
    assert(num_vertices(g) == num_vertices(h));
    assert(get_sorted_bundled_vertex_my_vertexes(g)
        == get_sorted_bundled_vertex_my_vertexes(h)
    );
}
```

This demonstration shows how the Markov chain is created using the ‘create_bundled_edges_and_vertices_markov_chain’ function (algorithm 191), saved and then loaded.

11.7 Load an undirected graph with bundled edges and vertices from a .dot file

When loading a graph from file, one needs to specify a type of graph. In this example, an undirected graph with bundled edges and vertices is loaded, as shown in algorithm 208:

Algorithm 208 Loading an undirected graph with bundled edges and vertices from a .dot file

```

#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "
    create_empty_undirected_bundled_edges_and_vertices_graph
    .h"
#include "is_regular_file.h"
#include "graphviz_decode.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    my_bundled_vertex,
    my_bundled_edge
>
load_undirected_bundled_edges_and_vertices_graph_from_dot
(
    const std::string& dot_filename
)
{
    assert(is_regular_file(dot_filename));
    std::ifstream f(dot_filename.c_str());
    auto g =
        create_empty_undirected_bundled_edges_and_vertices_graph
        ();

    boost::dynamic_properties dp(boost::
        ignore_other_properties);
    dp.property("label", get(&my_bundled_vertex::m_name, g));
    ;
    dp.property("comment", get(&my_bundled_vertex::
        m_description, g));
    dp.property("width", get(&my_bundled_vertex::m_x, g));
    dp.property("height", get(&my_bundled_vertex::m_y, g));
    dp.property("edge_id", get(&my_bundled_edge::m_name, g));
    ;
    dp.property("label", get(&my_bundled_edge::m_name, g));
    dp.property("comment", get(&my_bundled_edge::
        m_description, g));
    dp.property("width", get(&my_bundled_edge::m_width, g));
    ;
    dp.property("height", get(&my_bundled_edge::m_height, g
        ));
    boost::read_graphviz(f, g, dp);

    //Decode vertices                230
    {
        const auto vip = vertices(g);
        const auto j = vip.second;
        for (auto i = vip.first; i!=j; ++i)
        {
            g[*i].m_name = graphviz_decode(g[*i].m_name);
            g[*i].m_description = graphviz_decode(g[*i].

```

The only difference with loading a directed graph, is that the initial empty graph is undirected instead. Chapter 11.6 describes the rationale of this function.

Algorithm 209 shows how to use the ‘load_undirected_bundled_vertices_graph_from_dot’ function:

Algorithm 209 Demonstration of the ‘load_undirected_bundled_edges_and_vertices_graph_from_dot’ function

```
#include "create_bundled_edges_and_vertices_k3_graph.h"
#include "get_sorted_bundled_vertex_my_vertexes.h"
#include "
    load_undirected_bundled_edges_and_vertices_graph_from_dot
    .h"
#include "save_bundled_edges_and_vertices_graph_to_dot.h"

void
load_undirected_bundled_edges_and_vertices_graph_from_dot_demo
() noexcept
{
    using boost::num_edges;
    using boost::num_vertices;

    const auto g
        = create_bundled_edges_and_vertices_k3_graph();
    const std::string filename{
        "create_bundled_edges_and_vertices_k3_graph.dot"
    };
    save_bundled_edges_and_vertices_graph_to_dot(g,
        filename);
    const auto h
        =
            load_undirected_bundled_edges_and_vertices_graph_from_dot
            (
                filename
            );
    assert(num_edges(g) == num_edges(h));
    assert(num_vertices(g) == num_vertices(h));
    assert(get_sorted_bundled_vertex_my_vertexes(g)
        == get_sorted_bundled_vertex_my_vertexes(h)
    );
}
```

This demonstration shows how K_2 with bundled vertices is created using the ‘create_bundled_vertices_k2_graph’ function (algorithm 223), saved and

then loaded. The loaded graph is checked to be a graph similar to the original.

12 Building graphs with custom vertices

Instead of using bundled properties, you can also add a new custom property. The difference is that instead of having a class *as* a vertex, vertices have *an additional property* where the ‘my_custom_vertex’ is stored, next to properties like vertex name, edge delay (see chapter 25.1 for all properties). The following graphs will be created:

- An empty directed graph that allows for custom vertices: see chapter 212
- An empty undirected graph that allows for custom vertices: see chapter 12.3
- A two-state Markov chain with custom vertices: see chapter 12.7
- K_2 with custom vertices: see chapter 12.8

In the process, some basic (sometimes bordering trivial) functions are shown:

- Installing a new vertex property, called ‘vertex_custom_type’: chapter 12.2
- Adding a custom vertex: see chapter 12.5
- Getting the custom vertices my_vertex-es: see chapter 12.6

These functions are mostly there for completion and showing which data types are used.

12.1 Creating the vertex class

Before creating an empty graph with custom vertices, that custom vertex class must be created. In this tutorial, it is called ‘my_custom_vertex’. ‘my_custom_vertex’ is a class that is nonsensical, but it can be replaced by any other class type.

Here I will show the header file of ‘my_custom_vertex’, as the implementation of it is not important:

Algorithm 210 Declaration of `my_custom_vertex`

```
#include <string>
#include <iosfwd>

class my_custom_vertex
{
public:
    explicit my_custom_vertex(
        const std::string& name = "",
        const std::string& description = "",
        const double x = 0.0,
        const double y = 0.0
    ) noexcept;
    const std::string& get_description() const noexcept;
    const std::string& get_name() const noexcept;
    double get_x() const noexcept;
    double get_y() const noexcept;
private:
    std::string m_name;
    std::string m_description;
    double m_x;
    double m_y;
};

bool operator==(const my_custom_vertex& lhs, const
    my_custom_vertex& rhs) noexcept;
bool operator!=(const my_custom_vertex& lhs, const
    my_custom_vertex& rhs) noexcept;
bool operator<(const my_custom_vertex& lhs, const
    my_custom_vertex& rhs) noexcept;

std::ostream& operator<<(std::ostream& os, const
    my_custom_vertex& v) noexcept;
std::istream& operator>>(std::istream& os,
    my_custom_vertex& v) noexcept;
```

‘`my_custom_vertex`’ is a class that has multiple properties:

- It has four private member variables: the double ‘`m_x`’ (‘`m_`’ stands for member), the double ‘`m_y`’, the `std::string m_name` and the `std::string m_description`. These variables are private, but there are getters supplied
- It has a default constructor
- It is copyable

- It is comparable for equality (it has operator==), which is needed for searching
- It can be streamed (it has both operator<< and operator>>), which is needed for file I/O.

Special characters like comma's, quotes and whitespace cannot be streamed without problems. The function 'graphviz_encode' (algorithm 335) can convert the elements to be streamed to a Graphviz-friendly version, which can be decoded by 'graphviz_decode' (algorithm 336).

12.2 Installing the new vertex property

Before creating an empty graph with custom vertices, this type must be installed as a vertex property. Installing a new property would have been easier, if 'more C++ compilers were standards conformant' ([8] chapter 3.6). Boost.Graph uses the BOOST_INSTALL_PROPERTY macro to allow using a custom property:

Algorithm 211 Installing the vertex_custom_type property

```
#include <boost/graph/properties.hpp>

namespace boost {
    enum vertex_custom_type_t { vertex_custom_type = 314 };
    BOOST_INSTALL_PROPERTY(vertex, custom_type);
}
```

The enum value 314 must be unique.

12.3 Create the empty directed graph with custom vertices

Algorithm 212 Creating an empty directed graph with custom vertices

```
#include <boost/graph/adjacency_list.hpp>
#include "install_vertex_custom_type.h"
#include "my_custom_vertex.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex
    >
>
>
create_empty_directed_custom_vertices_graph() noexcept
{
    return {};
}
```

This graph:

- has its out edges stored in a `std::vector` (due to the first `boost::vecS`)
- has its vertices stored in a `std::vector` (due to the second `boost::vecS`)
- is directed (due to the `boost::directedS`)
- The vertices have one property: they have a custom type, that is of data type `my_vertex` (due to the `boost::property< boost::vertex_custom_type_t, my_vertex>`)
- The edges and graph have no properties
- Edges are stored in a `std::list`

The `boost::adjacency_list` has a new, fourth template argument '`boost::property< boost::vertex_custom_type_t, my_vertex>`'. This can be read as: "vertices have the property '`boost::vertex_custom_type_t`', which is of data type '`my_vertex`'". Or simply: "vertices have a custom type called `my_vertex`".

The demo:

Algorithm 213 Demo how to create an empty directed graph with custom vertices

```
#include "create_empty_directed_custom_vertices_graph.h"

void create_empty_directed_custom_vertices_graph_demo()
    noexcept
{
    const auto g
        = create_empty_directed_custom_vertices_graph();
    assert(boost::num_edges(g) == 0);
    assert(boost::num_vertices(g) == 0);
}
```

12.4 Create the empty undirected graph with custom vertices

Algorithm 214 Creating an empty undirected graph with custom vertices

```
#include <boost/graph/adjacency_list.hpp>
#include "install_vertex_custom_type.h"
#include "my_custom_vertex.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex
    >
>
>
create_empty_undirected_custom_vertices_graph() noexcept
{
    return {};
}
```

This code is very similar to the code described in chapter 12.3, except that the directedness (the third template argument) is undirected (due to the `boost::undirectedS`).

The demo:

Algorithm 215 Demo how to create an empty undirected graph with custom vertices

```
#include "create_empty_undirected_custom_vertices_graph.h"

void create_empty_undirected_custom_vertices_graph_demo()
    noexcept
{
    const auto g
        = create_empty_undirected_custom_vertices_graph();
    assert(boost::num_edges(g) == 0);
    assert(boost::num_vertices(g) == 0);
}
```

12.5 Add a custom vertex

Adding a custom vertex is very similar to adding a named vertex (chapter 4.3).

Algorithm 216 Add a custom vertex

```
#include <type_traits>
#include <boost/graph/adjacency_list.hpp>
#include "install_vertex_custom_type.h"
// #include "my_custom_vertex.h"

template <typename graph, typename vertex_t>
typename boost::graph_traits<graph>::vertex_descriptor
add_custom_vertex(
    const vertex_t& v,
    graph& g
) noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const");

    const auto vd = boost::add_vertex(g);
    const auto my_custom_vertex_map
        = get(boost::vertex_custom_type, g);
    put(my_custom_vertex_map, vd, v);
    return vd;
}
```

When having added a new (abstract) vertex to the graph, the vertex de-

scriptor is used to set the my_vertex in the graph its my_vertex map (using 'get(boost::vertex_custom_type,g)').

Here is the demo:

Algorithm 217 Demo of 'add_custom_vertex'

```
#include <cassert>
#include "add_custom_vertex.h"
#include "create_empty_directed_custom_vertices_graph.h"
#include "create_empty_undirected_custom_vertices_graph.h"
"

void add_custom_vertex_demo() noexcept
{
    auto g
        = create_empty_directed_custom_vertices_graph();
    assert(boost::num_vertices(g) == 0);
    assert(boost::num_edges(g) == 0);
    add_custom_vertex(my_custom_vertex("X"), g);
    assert(boost::num_vertices(g) == 1);
    assert(boost::num_edges(g) == 0);

    auto h
        = create_empty_undirected_custom_vertices_graph();
    assert(boost::num_vertices(h) == 0);
    assert(boost::num_edges(h) == 0);
    add_custom_vertex(my_custom_vertex("X"), h);
    assert(boost::num_vertices(h) == 1);
    assert(boost::num_edges(h) == 0);
}
```

12.6 Getting the vertices' my_vertexes¹⁰

When the vertices of a graph have any associated my_vertex, one can extract these as such:

¹⁰the name 'my_vertexes' is chosen to indicate this function returns a container of my_vertex

Algorithm 218 Get the my_custom_vertex objects

```
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include "install_vertex_custom_type.h"
#include "my_custom_vertex.h"

template <typename graph>
std::vector<my_custom_vertex> get_my_custom_vertexes(
    const graph& g
) noexcept
{
    using vd = typename graph::vertex_descriptor;

    std::vector<my_custom_vertex> v(boost::num_vertices(g))
        ;
    const auto vip = vertices(g);
    std::transform(vip.first, vip.second, std::begin(v),
        [g](const vd& d)
        {
            const auto my_custom_vertexes_map
                = get(boost::vertex_custom_type, g
            );
            return get(my_custom_vertexes_map, d);
        }
    );
    return v;
}
```

The my_vertex object associated with the vertices are obtained from a boost::property_map and then put into a std::vector.

The order of the ‘my_custom_vertex’ objects may be different after saving and loading.

When trying to get the vertices’ my_vertex from a graph without my_vertex objects associated, you will get the error ‘formed reference to void’ (see chapter 24.1).

Demo:

Algorithm 219 Demo how to the vertices' my_custom_vertex objects

```
#include <cassert>
#include "create_custom_vertices_k2_graph.h"
#include "get_my_custom_vertexes.h"

void get_my_custom_vertexes_demo() noexcept
{
    const auto g = create_custom_vertices_k2_graph();
    const std::vector<my_custom_vertex>
        expected_my_custom_vertexes{
            my_custom_vertex("A","source",0.0,0.0),
            my_custom_vertex("B","target",3.14,3.14)
        };
    const std::vector<my_custom_vertex> vertexes{
        get_my_custom_vertexes(g)
    };
    assert(expected_my_custom_vertexes == vertexes);
}
```

12.7 Creating a two-state Markov chain with custom vertices

12.7.1 Graph

Figure 44 shows the graph that will be reproduced:

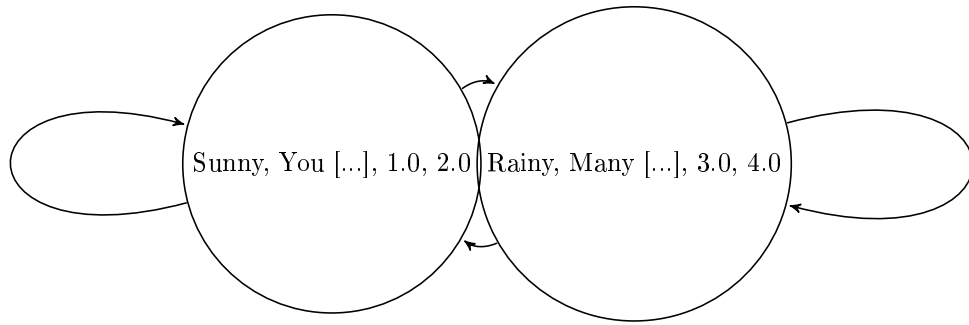


Figure 44: A two-state Markov chain where the vertices have custom properties and the edges have no properties. The vertices' properties are nonsensical

12.7.2 Function to create such a graph

Here is the code creating a two-state Markov chain with custom vertices:

Algorithm 220 Creating the two-state Markov chain as depicted in figure 44

```
#include <cassert>
#include "create_empty_directed_custom_vertices_graph.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex
    >
>
>
create_custom_vertices_markov_chain() noexcept
{
    auto g
        = create_empty_directed_custom_vertices_graph();
    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);
    const auto aer_aa = boost::add_edge(vd_a, vd_a, g);
    assert(aer_aa.second);
    const auto aer_ab = boost::add_edge(vd_a, vd_b, g);
    assert(aer_ab.second);
    const auto aer_ba = boost::add_edge(vd_b, vd_a, g);
    assert(aer_ba.second);
    const auto aer_bb = boost::add_edge(vd_b, vd_b, g);
    assert(aer_bb.second);

    auto my_custom_vertex_map = get(
        boost::vertex_custom_type, g
    );
    put(my_custom_vertex_map, vd_a,
        my_custom_vertex("Sunny", "Yellow_thing", 1.0, 2.0)
    );
    put(my_custom_vertex_map, vd_b,
        my_custom_vertex("Rainy", "Grey_things", 3.0, 4.0)
    );
    return g;
}
```

12.7.3 Creating such a graph

Here is the demo:

Algorithm 221 Demo of the ‘create_custom_vertices_markov_chain’ function (algorithm 220)

```
#include <cassert>
#include "create_custom_vertices_markov_chain.h"
#include "get_my_custom_vertexes.h"

void create_custom_vertices_markov_chain_demo() noexcept
{
    const auto g
        = create_custom_vertices_markov_chain();
    const std::vector<my_custom_vertex>
        expected_my_custom_vertexes{
            my_custom_vertex("Sunny", "Yellow_thing", 1.0, 2.0),
            my_custom_vertex("Rainy", "Grey_things", 3.0, 4.0)
        };
    const std::vector<my_custom_vertex>
        vertex_my_custom_vertexes{
            get_my_custom_vertexes(g)
        };
    assert(expected_my_custom_vertexes
        == vertex_my_custom_vertexes
    );
}
```

12.7.4 The .dot file produced

Algorithm 222 .dot file created from the ‘create_custom_vertices_markov_chain’ function (algorithm 220), converted from graph to .dot file using algorithm 241

```
digraph G {
0[label="Sunny,Yellow$$$SPACE$$$thing,1,1"];
1[label="Rainy,Grey$$$SPACE$$$things,3,3"];
0->0 ;
0->1 ;
1->0 ;
1->1 ;
}
```

This .dot file may look unexpectedly different: instead of a space, there is this ‘[[:SPACE:]]’ thing. This is because the function ‘graphviz_encode’ (algorithm 335) made this conversion. In this example, I could have simply surrounded the content by quotes, and this would have worked. I chose to use ‘graphviz_encode’

because it works in all contexts.

12.7.5 The .svg file produced

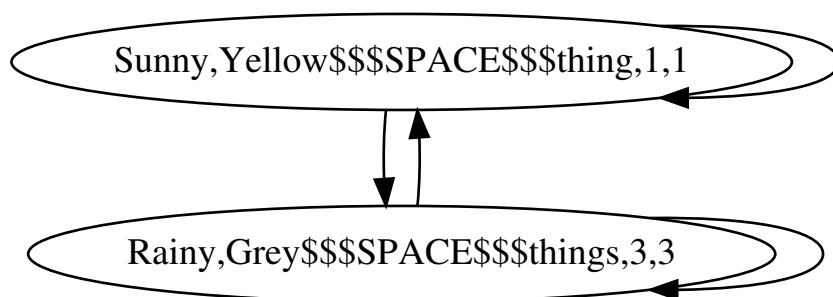


Figure 45: .svg file created from the ‘create_custom_vertices_markov_chain’ function (algorithm 220) its .dot file, converted from .dot file to .svg using algorithm 339

This .svg file may look unexpectedly different: instead of a space, there is this ‘[:SPACE:]’ thing. This is because the function ‘graphviz_encode’ (algorithm 335) made this conversion.

12.8 Creating K_2 with custom vertices

12.8.1 Graph

We reproduce the K_2 with named vertices of chapter 4.6 , but with our custom vertices instead.

12.8.2 Function to create such a graph

Algorithm 223 Creating K_2 as depicted in figure 18

```
#include "create_empty_undirected_custom_vertices_graph.h"
"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex
    >
>
>
create_custom_vertices_k2_graph() noexcept
{
    auto g = create_empty_undirected_custom_vertices_graph
        ();
    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);
    const auto aer = boost::add_edge(vd_a, vd_b, g);
    assert(aer.second);

    auto my_custom_vertex_map = get(
        boost::vertex_custom_type, g
    );
    put(my_custom_vertex_map, vd_a,
        my_custom_vertex("A", "source", 0.0, 0.0)
    );
    put(my_custom_vertex_map, vd_b,
        my_custom_vertex("B", "target", 3.14, 3.14)
    );

    return g;
}
```

Most of the code is a slight modification of the ‘create_named_vertices_k2_graph’ function (algorithm 65). In the end, the my_vertices are obtained as a boost::property_map and set with two custom my_vertex objects.

12.8.3 Creating such a graph

Demo:

Algorithm 224 Demo of the ‘create_custom_vertices_k2_graph’ function (algorithm 223)

```
#include <cassert>
#include <iostream>
#include "create_custom_vertices_k2_graph.h"
#include "has_custom_vertex_with_my_vertex.h"

void create_custom_vertices_k2_graph_demo() noexcept
{
    const auto g = create_custom_vertices_k2_graph();
    assert(boost::num_edges(g) == 1);
    assert(boost::num_vertices(g) == 2);
    assert(has_custom_vertex_with_my_custom_vertex(
        my_custom_vertex("A", "source", 0.0, 0.0), g)
    );
    assert(has_custom_vertex_with_my_custom_vertex(
        my_custom_vertex("B", "target", 3.14, 3.14), g)
    );
}
```

12.8.4 The .dot file produced

Algorithm 225 .dot file created from the ‘create_custom_vertices_k2_graph’ function (algorithm 223), converted from graph to .dot file using algorithm 48

```
graph G {
0[label="A,source,0,0"];
1[label="B,target,3.14,3.14"];
0--1 ;
}
```

12.8.5 The .svg file produced

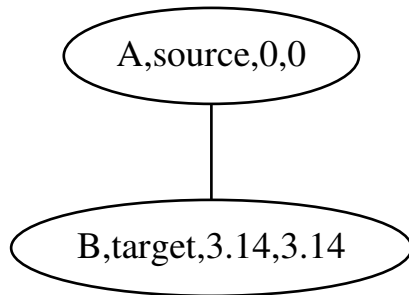


Figure 46: .svg file created from the ‘create_custom_vertices_k2_graph’ function (algorithm 223) its .dot file, converted from .dot file to .svg using algorithm 339

12.9 ► Creating a path graph with custom vertices

Here we create a path graph with custom vertices

12.9.1 Graph

Here I show a path graph with four vertices (see figure 47):

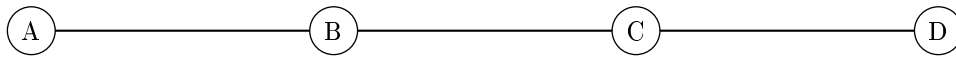


Figure 47: A path graph with four vertices

12.9.2 Function to create such a graph

To create a path graph, the following code can be used:

Algorithm 226 Creating a path graph as depicted in figure 47

```
#include <vector>
#include "add_custom_vertex.h"
#include "create_empty_undirected_custom_vertices_graph.h"
"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex
    >
>
>
create_custom_vertices_path_graph(
    const std::vector<my_custom_vertex>& names
) noexcept
{
    auto g = create_empty_undirected_custom_vertices_graph
        ();
    if (names.size() == 0) { return g; }
    auto vd_1 = add_custom_vertex(*names.begin(), g);
    if (names.size() == 1) return g;
    const auto j = std::end(names);
    auto i = std::begin(names);
    for (++i; i!=j; ++i) //Skip first
    {
        auto vd_2 = add_custom_vertex(*i, g);
        const auto aer = boost::add_edge(vd_1, vd_2, g);
        assert(aer.second);
        vd_1 = vd_2;
    }
    return g;
}
```

12.9.3 Creating such a graph

Algorithm 227 demonstrates how to create a path graph with named vertices and checks if it has the correct amount of edges and vertices:

Algorithm 227 Demonstration of ‘create_named_vertices_path_graph’

```
#include <cassert>
#include "create_custom_vertices_path_graph.h"

void create_custom_vertices_path_graph_demo() noexcept
{
    const auto g = create_custom_vertices_path_graph(
        {
            my_custom_vertex("A"),
            my_custom_vertex("B"),
            my_custom_vertex("C")
        }
    );
    assert(boost::num_edges(g) == 2);
    assert(boost::num_vertices(g) == 3);
}
```

12.9.4 The .dot file produced

This graph can be converted to the .dot file as shown in algorithm 228:

Algorithm 228 .dot file created from the ‘create_named_vertices_path_graph’ function (algorithm 226), converted from graph to .dot file using algorithm 48

12.9.5 The .svg file produced

The .dot file can be converted to the .svg as shown in figure 48:

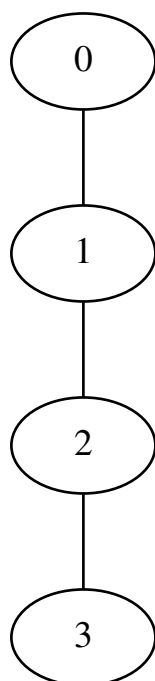


Figure 48: .svg file created from the ‘create_named_vertices_path_graph’ function (algorithm 226) its .dot file, converted from .dot file to .svg using algorithm 339

13 Working on graphs with custom vertices (as a custom property)

When using graphs with custom vertices, their state gives a way to find a vertex and working with it. This chapter shows some basic operations on graphs with custom vertices.

- Check if there exists a vertex with a certain ‘my_vertex’: chapter 13.1
- Find a vertex with a certain ‘my_vertex’: chapter 13.2
- Get a vertex its ‘my_vertex’ from its vertex descriptor: chapter 13.3
- Set a vertex its ‘my_vertex’ using its vertex descriptor: chapter 13.4
- Setting all vertices their ‘my_vertex’es: chapter 13.5
- Storing an directed/undirected graph with custom vertices as a .dot file: chapter 13.7

- Loading a directed graph with custom vertices from a .dot file: chapter 13.8
- Loading an undirected directed graph with custom vertices from a .dot file: chapter 13.9

13.1 Has a custom vertex with a my_vertex

Before modifying our vertices, let's first determine if we can find a vertex by its custom type ('my_vertex') in a graph. After obtaining a my_vertex map, we obtain the vertex iterators, dereference these to obtain the vertex descriptors and then compare each vertex its my_vertex with the one desired.

Algorithm 229 Find if there is vertex with a certain my_vertex

```
#include <string>
#include <boost/graph/properties.hpp>
#include "install_vertex_custom_type.h"
#include "my_custom_vertex.h"

template <typename graph>
bool has_custom_vertex_with_my_custom_vertex(
    const my_custom_vertex& v,
    const graph& g
) noexcept
{
    using vd = typename graph::vertex_descriptor;

    const auto vip = vertices(g);
    return std::find_if(vip.first, vip.second,
        [v, g](const vd& d)
        {
            const auto my_custom_vertexes_map
                = get(boost::vertex_custom_type, g);
            return get(my_custom_vertexes_map, d) == v;
        }
    ) != vip.second;
}
```

This function can be demonstrated as in algorithm 230, where a certain my_vertex cannot be found in an empty graph. After adding the desired my_vertex, it is found.

Algorithm 230 Demonstration of the ‘has_custom_vertex_with_my_vertex’ function

```
#include <cassert>
#include <iostream>

#include "add_custom_vertex.h"
#include "create_empty_undirected_custom_vertices_graph.h"
#include "has_custom_vertex_with_my_vertex.h"
#include "install_vertex_custom_type.h"
#include "my_custom_vertex.h"

void has_custom_vertex_with_my_custom_vertex_demo()
    noexcept
{
    auto g = create_empty_undirected_custom_vertices_graph
        ();
    assert(!has_custom_vertex_with_my_custom_vertex(
        my_custom_vertex("Felix"), g));
    add_custom_vertex(my_custom_vertex("Felix"), g);
    assert(has_custom_vertex_with_my_custom_vertex(
        my_custom_vertex("Felix"), g));
}
```

Note that this function only finds if there is at least one custom vertex with that my_vertex: it does not tell how many custom vertices with that my_vertex exist in the graph.

13.2 Find a custom vertex with a certain my_vertex

Where STL functions work with iterators, here we obtain a vertex descriptor (see chapter 2.6) to obtain a handle to the desired vertex. Algorithm 231 shows how to obtain a vertex descriptor to the first vertex found with a specific my_vertex value.

Algorithm 231 Find the first vertex with a certain `my_vertex`

```
#include <cassert>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "has_custom_vertex_with_my_vertex.h"
#include "install_vertex_custom_type.h"
#include "my_custom_vertex.h"

template <typename graph>
typename boost::graph_traits<graph>::vertex_descriptor
find_first_custom_vertex_with_my_vertex(
    const my_custom_vertex& v,
    const graph& g
) noexcept
{
    using vd = typename graph::vertex_descriptor;
    const auto vip = vertices(g);
    const auto i = std::find_if(
        vip.first, vip.second,
        [v,g](const vd d) {
            const auto my_vertex_map = get(boost::
                vertex_custom_type, g);
            return get(my_vertex_map, d) == v;
        }
    );
    assert(i != vip.second);
    return *i;
}
```

With the vertex descriptor obtained, one can read and modify the vertex and the edges surrounding it. Algorithm 232 shows some examples of how to do so.

Algorithm 232 Demonstration of the ‘find_first_custom_vertex_with_my_vertex’ function

```
#include <cassert>

#include "create_custom_vertices_k2_graph.h"
#include "find_first_custom_vertex_with_my_vertex.h"

void find_first_custom_vertex_with_my_vertex_demo()
    noexcept
{
    const auto g = create_custom_vertices_k2_graph();
    const auto vd = find_first_custom_vertex_with_my_vertex
        (
            my_custom_vertex("A", "source", 0.0, 0.0),
            g
        );
    assert(out_degree(vd, g) == 1);
    assert(in_degree(vd, g) == 1);
}
```

13.3 Get a custom vertex its my_vertex

To obtain the name from a vertex descriptor, one needs to pull out the my_vertexes¹¹ map and then look up the vertex of interest.

¹¹Bad English intended: my_vertexes = multiple my_vertex objects, vertices = multiple graph nodes

Algorithm 233 Get a `my_custom_vertex` its `my_vertex` from its vertex descriptor

```
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "install_vertex_custom_type.h"
#include "my_custom_vertex.h"

template <typename graph>
my_custom_vertex get_my_custom_vertex(
    const typename boost::graph_traits<graph>::
        vertex_descriptor& vd,
    const graph& g
) noexcept
{
    const auto my_custom_vertexes_map
        = get(boost::vertex_custom_type,
            g
        );
    return get(my_custom_vertexes_map, vd);
}
```

To use ‘`get_my_custom_vertex`’, one first needs to obtain a vertex descriptor. Algorithm 234 shows a simple example.

Algorithm 234 Demonstration if the ‘`get_my_custom_vertex`’ function

```
#include <cassert>
#include "add_custom_vertex.h"
#include "create_empty_undirected_custom_vertices_graph.h"
#include "find_first_custom_vertex_with_my_vertex.h"
#include "get_my_custom_vertex.h"

void get_my_custom_vertex_demo() noexcept
{
    auto g = create_empty_undirected_custom_vertices_graph
        ();
    const my_custom_vertex name{"Dex"};
    add_custom_vertex(name, g);
    const auto vd
        = find_first_custom_vertex_with_my_vertex(name, g);
    assert(get_my_custom_vertex(vd, g) == name);
}
```

13.4 Set a custom vertex its my_vertex

If you know how to get the my_vertex from a vertex descriptor, setting it is just as easy, as shown in algorithm 235.

Algorithm 235 Set a custom vertex its my_vertex from its vertex descriptor

```
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "install_vertex_custom_type.h"
#include "my_custom_vertex.h"

template <typename graph>
void set_my_custom_vertex(
    const my_custom_vertex& v,
    const typename boost::graph_traits<graph>::
        vertex_descriptor& vd,
    graph& g
) noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const"
    );

    const auto my_custom_vertexes_map
        = get(boost::vertex_custom_type, g);
    put(my_custom_vertexes_map, vd, v);
}
```

To use ‘set_my_custom_vertex’, one first needs to obtain a vertex descriptor. Algorithm 236 shows a simple example.

Algorithm 236 Demonstration if the ‘set_my_custom_vertex’ function

```
#include <cassert>

#include "add_custom_vertex.h"
#include "create_empty_undirected_custom_vertices_graph.h"
"
#include "find_first_custom_vertex_with_my_vertex.h"
#include "get_my_custom_vertex.h"
#include "set_my_custom_vertex.h"

void set_my_custom_vertex_demo() noexcept
{
    auto g
        = create_empty_undirected_custom_vertices_graph();
    const my_custom_vertex old_vertex{"Dex"};
    add_custom_vertex(old_vertex, g);
    const auto vd
        = find_first_custom_vertex_with_my_vertex(old_vertex,
            g);
    assert(get_my_custom_vertex(vd, g)
        == old_vertex
    );
    const my_custom_vertex new_vertex{"Diggy"};
    set_my_custom_vertex(
        new_vertex, vd, g
    );
    assert(get_my_custom_vertex(vd, g)
        == new_vertex
    );
}
```

13.5 Setting all custom vertices’ my_vertex objects

When the vertices of a graph are associated with my_vertex objects, one can set these my_vertexes as such:

Algorithm 237 Setting the custom vertices' my_vertexes

```
#include <string>
#include <vector>

#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>

#include "install_vertex_custom_type.h"
#include "my_custom_vertex.h"

template <typename graph>
void set_my_custom_vertexes(
    graph& g,
    const std::vector<my_custom_vertex>& my_custom_vertexes
) noexcept
{
    static_assert(!std::is_const<graph>::value, "graph_
        cannot_be_const");

    const auto my_custom_vertex_map
        = get(boost::vertex_custom_type, g);

    auto my_custom_vertexes_begin = std::begin(
        my_custom_vertexes);
    const auto my_custom_vertexes_end = std::end(
        my_custom_vertexes);
    const auto vip = vertices(g);
    const auto j = vip.second;
    for (
        auto i = vip.first;
        i!=j; ++i,
        ++my_custom_vertexes_begin
    ) {
        assert(my_custom_vertexes_begin !=
            my_custom_vertexes_end);
        put(my_custom_vertex_map, *i, *
            my_custom_vertexes_begin);
    }
}
```

An impressive feature is that getting the property map holding the graph its names is not a copy, but a reference. Otherwise, modifying 'my_vertexes_map' (obtained by non-reference) would only modify a copy.

13.6 ► Are two graphs with custom vertices isomorphic?

Algorithm 5.14 checked if two graphs with named vertices are ‘label isomorphic’. Here, we do the same for custom vertices.

To do this, there are two steps needed:

1. Map all `my_custom_vertex` objects to an unsigned int.
2. Compare the two graphs with that map

Below the class ‘`my_custom_vertex_invariant`’ is shown. Its `std::map` maps the vertex names to an unsigned integer, which is done in the member function ‘`collect_names`’. The purpose of this, is that it is easier to compare integers than custom vertices. Note that `operator<` must be implemented for the custom class for this to compile.

Algorithm 238 The ‘custom_vertex_invariant’ functor

```
#include <map>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/isomorphism.hpp>
#include "my_custom_vertex.h"
#include "install_vertex_custom_type.h"

template <class graph>
struct custom_vertex_invariant {
    using custom_vertex_to_int_map = std::map<
        my_custom_vertex, size_t>;
    using result_type = size_t;
    using argument_type = typename graph::vertex_descriptor
        ;

    const graph& m_graph;
    custom_vertex_to_int_map& m_mappings;

    size_t operator()(argument_type u) const {
        return m_mappings.at(boost::get(boost::
            vertex_custom_type, m_graph, u));
    }
    size_t max() const noexcept { return m_mappings.size();
    }

    void collect_custom() noexcept {
        for (const auto vd : boost::make_iterator_range(boost
            ::vertices(m_graph))) {
            const size_t next_id = m_mappings.size();
            const auto ins = m_mappings.insert(
                { boost::get(boost::vertex_custom_type, m_graph,
                    vd), next_id }
            );
            if (ins.second) {
                //std::cout << "Mapped '" << ins.first->first <<
                    "' to " << ins.first->second << "\n";
            }
        }
    }
};
```

To check for ‘custom vertexness isomorphism’, multiple things need to be put in place for ‘boost::isomorphism’ to work with:

Algorithm 239 Check if two graphs with custom vertices are isomorphic

```
#include "custom_vertex_invariant.h"

#include <boost/graph/vf2_sub_graph_iso.hpp>
#include <boost/graph/graph_utility.hpp>

template <typename graph>
bool is_custom_vertices_isomorphic(
    const graph &g,
    const graph &h
) noexcept {
    using vd = typename graph::vertex_descriptor;
    auto vertex_index_map = get(boost::vertex_index, g);
    std::vector<vd> iso(boost::num_vertices(g));

    typename custom_vertex_invariant<graph>::
        custom_vertex_to_int_map shared_custom;
    custom_vertex_invariant<graph> inv1{g, shared_custom};
    custom_vertex_invariant<graph> inv2{h, shared_custom};
    inv1.collect_custom();
    inv2.collect_custom();

    return boost::isomorphism(g, h,
        boost::isomorphism_map(
            make_iterator_property_map(
                iso.begin(),
                vertex_index_map
            )
        )
        .vertex_invariant1(inv1)
        .vertex_invariant2(inv2)
    );
}
```

This demonstration code creates three path graphs, of which two are ‘label isomorphic’:

Algorithm 240 Demo of the ‘is_named_vertices_isomorphic’ function

```
#include <cassert>
#include "create_custom_vertices_path_graph.h"
#include "is_custom_vertices_isomorphic.h"

void is_custom_vertices_isomorphic_demo() noexcept
{
    const auto g = create_custom_vertices_path_graph(
        {
            my_custom_vertex("Alpha"),
            my_custom_vertex("Beta"),
            my_custom_vertex("Gamma")
        }
    );
    const auto h = create_custom_vertices_path_graph(
        {
            my_custom_vertex("Gamma"),
            my_custom_vertex("Beta"),
            my_custom_vertex("Alpha")
        }
    );
    const auto i = create_custom_vertices_path_graph(
        {
            my_custom_vertex("Alpha"),
            my_custom_vertex("Gamma"),
            my_custom_vertex("Beta")
        }
    );
    assert( is_custom_vertices_isomorphic(g,h) );
    assert( !is_custom_vertices_isomorphic(g,i) );
}
```

13.7 Storing a graph with custom vertices as a .dot

If you used the `create_custom_vertices_k2_graph` function (algorithm 223) to produce a K_2 graph with vertices associated with `my_vertex` objects, you can store these `my_vertexes` additionally with algorithm 241:

Algorithm 241 Storing a graph with custom vertices as a .dot file

```
#include <fstream>
#include <string>
#include <boost/graph/graphviz.hpp>
#include <boost/graph/properties.hpp>
#include "get_my_custom_vertexes.h"

template <typename graph>
void save_custom_vertices_graph_to_dot(
    const graph& g,
    const std::string& filename
) noexcept
{
    using vd = typename graph::vertex_descriptor;

    std::ofstream f(filename);
    boost::write_graphviz(
        f,
        g,
        [g](std::ostream& out, const vd& v) {
            const auto my_custom_vertexes_map
                = get(boost::vertex_custom_type, g)
            ;
            const my_custom_vertex m{get(my_custom_vertexes_map, v)};
            out << "[label=\"" << m << "\"]";
        }
    );
}
```

13.8 Loading a directed graph with custom vertices from a .dot

When loading a graph from file, one needs to specify a type of graph. In this example, an directed graph with custom vertices is loaded, as shown in algorithm 242:

Algorithm 242 Loading a directed graph with custom vertices from a .dot file

```
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "create_empty_directed_custom_vertices_graph.h"
#include "is_regular_file.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex
    >
>
>
load_directed_custom_vertices_graph_from_dot(
    const std::string& dot_filename
)
{
    assert(is_regular_file(dot_filename));
    std::ifstream f(dot_filename.c_str());
    auto g = create_empty_directed_custom_vertices_graph();
    boost::dynamic_properties dp(boost::
        ignore_other_properties);
    dp.property("label", get(boost::vertex_custom_type, g))
        ;
    boost::read_graphviz(f, g, dp);
    return g;
}
```

In this algorithm, first it is checked if the file to load exists. Then an empty directed graph is created. Next to this, a `boost::dynamic_properties` is created with its default constructor, after which we direct the `boost::dynamic_properties` to find a ‘node_id’ and ‘label’ in the vertex name map, ‘edge_id’ and ‘label’ to the edge name map. From this and the empty graph, ‘`boost::read_graphviz`’ is called to build up the graph.

Algorithm 243 shows how to use the ‘`load_directed_custom_vertices_graph_from_dot`’ function:

Algorithm 243 Demonstration of the ‘load_directed_custom_vertices_graph_from_dot’ function

```
#include "create_custom_vertices_markov_chain.h"
#include "load_directed_custom_vertices_graph_from_dot.h"
#include "save_custom_vertices_graph_to_dot.h"
#include "get_my_custom_vertexes.h"

void load_directed_custom_vertices_graph_from_dot_demo()
    noexcept
{
    using boost::num_edges;
    using boost::num_vertices;

    const auto g
        = create_custom_vertices_markov_chain();
    const std::string filename{
        "create_custom_vertices_markov_chain.dot"
    };
    save_custom_vertices_graph_to_dot(g, filename);
    const auto h
        = load_directed_custom_vertices_graph_from_dot(
            filename);
    assert(num_edges(g) == num_edges(h));
    assert(num_vertices(g) == num_vertices(h));
    assert(get_my_custom_vertexes(g)
        == get_my_custom_vertexes(h)
    );
}
```

This demonstration shows how the Markov chain is created using the ‘create_custom_vertices_markov_chain’ function (algorithm 220), saved and then loaded. The loaded graph is then checked to be identical to the original.

13.9 Loading an undirected graph with custom vertices from a .dot

When loading a graph from file, one needs to specify a type of graph. In this example, an undirected graph with custom vertices is loaded, as shown in algorithm 244:

Algorithm 244 Loading an undirected graph with custom vertices from a .dot file

```
#include <fstream>
#include <boost/graph/graphviz.hpp>

#include "create_empty_undirected_custom_vertices_graph.h"
#include "is_regular_file.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex
    >
>
>
load_undirected_custom_vertices_graph_from_dot(
    const std::string& dot_filename
)
{
    assert(is_regular_file(dot_filename));
    std::ifstream f(dot_filename.c_str());
    auto g = create_empty_undirected_custom_vertices_graph
        ();
    boost::dynamic_properties dp(boost::
        ignore_other_properties);
    dp.property("label", get(boost::vertex_custom_type, g))
        ;
    boost::read_graphviz(f, g, dp);
    return g;
}
```

The only difference with loading a directed graph, is that the initial empty graph is undirected instead. Chapter 13.8 describes the rationale of this function.

Algorithm 245 shows how to use the ‘load_undirected_custom_vertices_graph_from_dot’ function:

Algorithm 245 Demonstration of the ‘load_undirected_custom_vertices_graph_from_dot’ function

```
#include <cassert>
#include "create_custom_vertices_k2_graph.h"
#include "load_undirected_custom_vertices_graph_from_dot.h"
#include "save_custom_vertices_graph_to_dot.h"
#include "get_my_custom_vertexes.h"

void load_undirected_custom_vertices_graph_from_dot_demo
() noexcept
{
    using boost::num_edges;
    using boost::num_vertices;

    const auto g
        = create_custom_vertices_k2_graph();
    const std::string filename{
        "create_custom_vertices_k2_graph.dot"
    };
    save_custom_vertices_graph_to_dot(g, filename);
    const auto h
        = load_undirected_custom_vertices_graph_from_dot(
            filename);
    assert(num_edges(g) == num_edges(h));
    assert(num_vertices(g) == num_vertices(h));
    assert(get_my_custom_vertexes(g) ==
        get_my_custom_vertexes(h));
}
```

This demonstration shows how K_2 with custom vertices is created using the ‘create_custom_vertices_k2_graph’ function (algorithm 223), saved and then loaded. The loaded graph is then checked to be identical to the original.

14 Building graphs with custom and selectable vertices

We have added one custom vertex property, here we add a second: if the vertex is selected.

- An empty directed graph that allows for custom and selectable vertices: see chapter 14.2
- An empty undirected graph that allows for custom and selectable vertices:

see chapter 14.3

- A two-state Markov chain with custom and selectable vertices: see chapter 14.5
- K_3 with custom and selectable vertices: see chapter 14.6

In the process, some basic (sometimes bordering trivial) functions are shown:

- Installing the new edge property: see chapter 14.1
- Adding a custom and selectable vertex: see chapter 14.4

These functions are mostly there for completion and showing which data types are used.

14.1 Installing the new `is_selected` property

Installing a new property would have been easier, if ‘more C++ compilers were standards conformant’ ([8], chapter 3.6, footnote at page 52). Boost.Graph uses the `BOOST_INSTALL_PROPERTY` macro to allow using a custom property:

Algorithm 246 Installing the `vertex_is_selected` property

```
#include <boost/graph/properties.hpp>

namespace boost {
    enum vertex_is_selected_t { vertex_is_selected = 31416 };
    BOOST_INSTALL_PROPERTY(vertex, is_selected);
}
```

The enum value 31415 must be unique.

14.2 Create an empty directed graph with custom and selectable vertices

Algorithm 247 Creating an empty directed graph with custom and selectable vertices

```
#include <boost/graph/adjacency_list.hpp>
#include "install_vertex_custom_type.h"
#include "install_vertex_is_selected.h"
#include "my_custom_vertex.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex,
        boost::property<
            boost::vertex_is_selected_t, bool
        >
    >
>
>
>
create_empty_directed_custom_and_selectable_vertices_graph
    () noexcept
{
    return {};
}
```

This code is very similar to the code described in chapter 12.3, except that there is a new, fourth template argument:

```
boost::property<boost::vertex_custom_type_t, my_custom_vertex,
    boost::property<boost::vertex_is_selected_t, bool,
>
```

This can be read as: “vertices have two properties: an associated custom type (of type `my_custom_vertex`) and an associated `is_selected` property (of type `bool`)”.

Demo:

Algorithm 248 Demonstration of the ‘create_empty_directed_custom_and_selectable_vertices_graph’ function

```
#include "
    create_empty_directed_custom_and_selectable_vertices_graph
    .h"

void
    create_empty_directed_custom_and_selectable_vertices_graph_demo
    () noexcept
{
    const auto g
        =
            create_empty_directed_custom_and_selectable_vertices_graph
            ();
    assert(boost::num_edges(g) == 0);
    assert(boost::num_vertices(g) == 0);
}
```

14.3 Create an empty undirected graph with custom and selectable vertices

Algorithm 249 Creating an empty undirected graph with custom and selectable vertices

```
#include <boost/graph/adjacency_list.hpp>
#include "install_vertex_custom_type.h"
#include "install_vertex_is_selected.h"
#include "my_custom_vertex.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex,
        boost::property<
            boost::vertex_is_selected_t, bool
        >
    >
>
>
>
create_empty_undirected_custom_and_selectable_vertices_graph
    () noexcept
{
    return {};
}
```

This code is very similar to the code described in chapter 14.2, except that the directedness (the third template argument) is undirected (due to the `boost::undirectedS`).

Demo:

Algorithm 250 Demonstration of the ‘create_empty_undirected_custom_and_selectable_vertices_graph’ function

```
#include "
    create_empty_undirected_custom_and_selectable_vertices_graph
    .h"

void
    create_empty_undirected_custom_and_selectable_vertices_graph_demo
    () noexcept
{
    const auto g
        =
            create_empty_undirected_custom_and_selectable_vertices_graph
            ();
    assert(boost::num_edges(g) == 0);
    assert(boost::num_vertices(g) == 0);
}
```

14.4 Add a custom and selectable vertex

Adding a custom and selectable vertex is very similar to adding a custom vertex (chapter 12.5).

Algorithm 251 Add a custom and selectable vertex

```
#include <type_traits>
#include <boost/graph/adjacency_list.hpp>
#include "install_vertex_custom_type.h"
#include "install_vertex_is_selected.h"
//#include "my_custom_vertex.h"

template <typename graph, typename vertex_t>
typename boost::graph_traits<graph>::vertex_descriptor
add_custom_and_selectable_vertex(
    const vertex_t& v,
    const bool is_selected,
    graph& g
) noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const");
};

const auto vd = boost::add_vertex(g);

const auto my_custom_vertex_map
    = get(boost::vertex_custom_type,
        g
    );
put(my_custom_vertex_map, vd, v);

const auto is_selected_map
    = get(boost::vertex_is_selected,
        g
    );
put(is_selected_map, vd, is_selected);
return vd;
}
```

When having added a new (abstract) vertex to the graph, the vertex descriptor is used to set the `my_custom_vertex` and the selectedness in the graph its `my_custom_vertex` and `is_selected` map.

Here is the demo:

Algorithm 252 Demo of ‘add_custom_and_selectable_vertex’

```
#include <cassert>
#include "add_custom_and_selectable_vertex.h"
#include "
    create_empty_directed_custom_and_selectable_vertices_graph
    .h"
#include "
    create_empty_undirected_custom_and_selectable_vertices_graph
    .h"

void add_custom_and_selectable_vertex_demo() noexcept
{
    auto g
        =
            create_empty_directed_custom_and_selectable_vertices_graph
            ();
    assert(boost::num_vertices(g) == 0);
    assert(boost::num_edges(g) == 0);
    add_custom_and_selectable_vertex(
        my_custom_vertex("X"),
        true,
        g
    );
    assert(boost::num_vertices(g) == 1);
    assert(boost::num_edges(g) == 0);

    auto h
        =
            create_empty_undirected_custom_and_selectable_vertices_graph
            ();
    assert(boost::num_vertices(h) == 0);
    assert(boost::num_edges(h) == 0);
    add_custom_and_selectable_vertex(
        my_custom_vertex("X"),
        false,
        h
    );
    assert(boost::num_vertices(h) == 1);
    assert(boost::num_edges(h) == 0);
}
```

14.5 Creating a Markov-chain with custom and selectable vertices

14.5.1 Graph

Figure 49 shows the graph that will be reproduced:

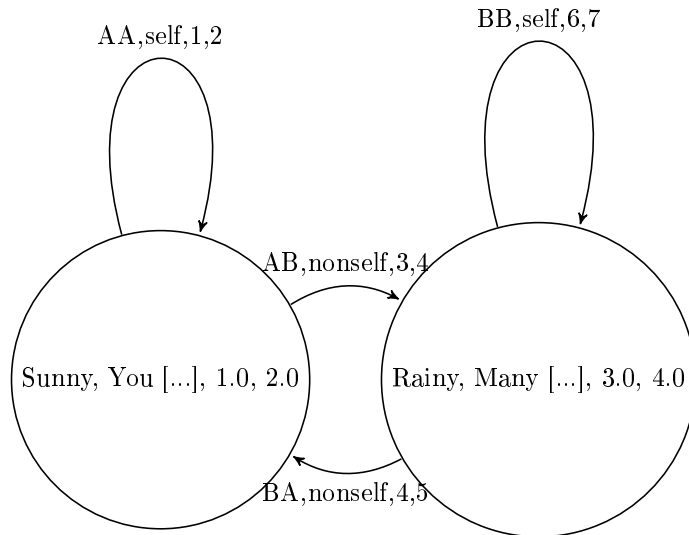


Figure 49: A two-state Markov chain where the edges and vertices have custom properties. The edges' and vertices' properties are nonsensical

14.5.2 Function to create such a graph

Here is the code creating a two-state Markov chain with custom edges and vertices:

Algorithm 253 Creating the two-state Markov chain as depicted in figure 49

```
#include <cassert>
#include "
    create_empty_directed_custom_and_selectable_vertices_graph
    .h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex,
        boost::property<
            boost::vertex_is_selected_t, bool
        >
    >
>
>
create_custom_and_selectable_vertices_markov_chain()
    noexcept
{
    auto g
        =
            create_empty_directed_custom_and_selectable_vertices_graph
                ();
    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);
    const auto aer_aa = boost::add_edge(vd_a, vd_a, g);
    assert(aer_aa.second);
    const auto aer_ab = boost::add_edge(vd_a, vd_b, g);
    assert(aer_ab.second);
    const auto aer_ba = boost::add_edge(vd_b, vd_a, g);
    assert(aer_ba.second);
    const auto aer_bb = boost::add_edge(vd_b, vd_b, g);
    assert(aer_bb.second);

    auto my_custom_vertex_map = get(
        boost::vertex_custom_type, g
    );
    put(my_custom_vertex_map, vd_a,
        my_custom_vertex("Sunny", "Yellow_thing", 1.0, 2.0)
    );
    put(my_custom_vertex_map, vd_b,
        my_custom_vertex("Rainy", "Grey_things", 3.0, 4.0)
    );
    auto is_selected_map = get(
        boost::vertex_is_selected, g
    );
    put(is_selected_map, vd_a, 275true);
    put(is_selected_map, vd_b, false);
    return g;
}
```

14.5.3 Creating such a graph

Here is the demo:

Algorithm 254 Demo of the ‘create_custom_and_selectable_vertices_markov_chain’ function (algorithm 253)

```
#include <cassert>
#include "
    create_custom_and_selectable_vertices_markov_chain.h"
#include "get_vertex_selectednesses.h"

void
    create_custom_and_selectable_vertices_markov_chain_demo
    () noexcept
{
    const auto g
        = create_custom_and_selectable_vertices_markov_chain
        ();
    const std::vector<bool>
        expected_selectednesses{
            true, false
        };
    const std::vector<bool>
        vertex_selectednesses{
            get_vertex_selectednesses(g)
        };
    assert(expected_selectednesses
        == vertex_selectednesses
    );
}
```

14.5.4 The .dot file produced

Algorithm 255 .dot file created from the 'create_custom_and_selectable_vertices_markov_chain' function (algorithm 253), converted from graph to .dot file using algorithm 48

```
digraph G {
0[label="Sunny,Yellow$$$SPACE$$$thing,1,1", regular="1"];
1[label="Rainy,Grey$$$SPACE$$$things,3,3", regular="0"];
0->0 ;
0->1 ;
1->0 ;
1->1 ;
}
```

14.5.5 The .svg file produced

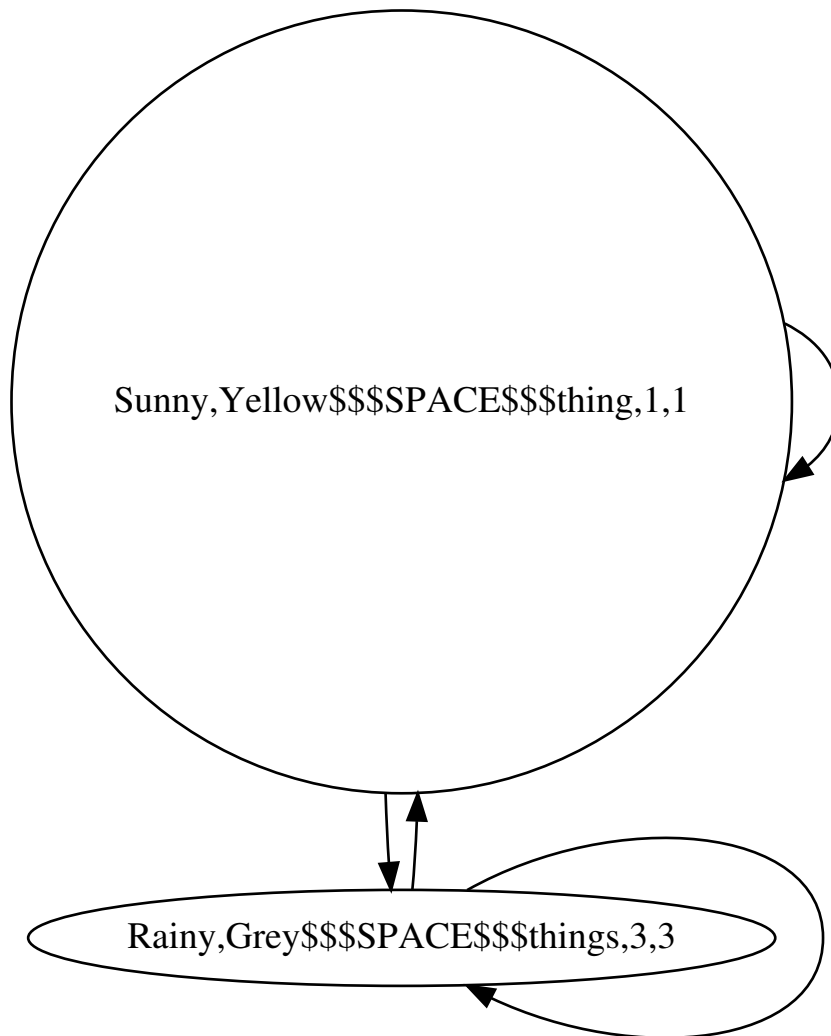


Figure 50: .svg file created from the ‘create_custom_and_selectable_vertices_markov_chain’ function (algorithm 220) its .dot file, converted from .dot file to .svg using algorithm 339

Note how the .svg changed its appearance due to the Graphviz ‘regular’ property (see chapter 25.2): the vertex labeled ‘Sunny’ is drawn according to the Graphviz ‘regular’ attribute, which makes it a circle. The other vertex, labeled ‘Rainy’ is not drawn as such and retained its ellipsoid appearance.

14.6 Creating K_2 with custom and selectable vertices

14.6.1 Graph

We reproduce the K_2 with custom vertices of chapter 12.8 , but now are vertices can be selected as well:

[graph here]

14.6.2 Function to create such a graph

Algorithm 256 Creating K_3 as depicted in figure 30

```
#include "
    create_empty_undirected_custom_and_selectable_vertices_graph
    .h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex,
        boost::property<
            boost::vertex_is_selected_t, bool
        >
    >
>
>
>
create_custom_and_selectable_vertices_k2_graph() noexcept
{
    auto g
        =
            create_empty_undirected_custom_and_selectable_vertices_graph
            ();
    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);
    const auto aer = boost::add_edge(vd_a, vd_b, g);
    assert(aer.second);

    auto my_custom_vertexes_map = get(
        boost::vertex_custom_type,
        g
    );
    put(my_custom_vertexes_map, vd_a,
        my_custom_vertex("A", "source", 0.0, 0.0)
    );
    put(my_custom_vertexes_map, vd_b,
        my_custom_vertex("B", "target", 3.14, 3.14)
    );

    auto is_selected_map = get(
        boost::vertex_is_selected,
        g
    );
    put(is_selected_map, vd_a, true);
    put(is_selected_map, vd_b, false);
    return g;
}
281
```

Most of the code is a slight modification of algorithm 223. In the end, the associated `my_custom_vertex` and `is_selected` properties are obtained as `boost::property_maps` and set with the desired `my_custom_vertex` objects and `selectednesses`.

14.6.3 Creating such a graph

Here is the demo:

Algorithm 257 Demo of the ‘`create_custom_and_selectable_vertices_k2_graph`’ function (algorithm 256)

```
#include <cassert>
#include "create_custom_and_selectable_vertices_k2_graph.h"
#include "has_custom_vertex_with_my_vertex.h"

void create_custom_and_selectable_vertices_k2_graph_demo
() noexcept
{
    const auto g =
        create_custom_and_selectable_vertices_k2_graph();
    assert(boost::num_edges(g) == 1);
    assert(boost::num_vertices(g) == 2);
    assert(has_custom_vertex_with_my_custom_vertex(
        my_custom_vertex("A", "source", 0.0, 0.0), g)
    );
    assert(has_custom_vertex_with_my_custom_vertex(
        my_custom_vertex("B", "target", 3.14, 3.14), g)
    );
}
```

14.6.4 The .dot file produced

Algorithm 258 .dot file created from the ‘`create_custom_and_selectable_vertices_k2_graph`’ function (algorithm 256), converted from graph to .dot file using algorithm 48

```
graph G {
0[label="A,source,0,0", regular="1"];
1[label="B,target,3.14,3.14", regular="0"];
0--1 ;
}
```

14.6.5 The .svg file produced

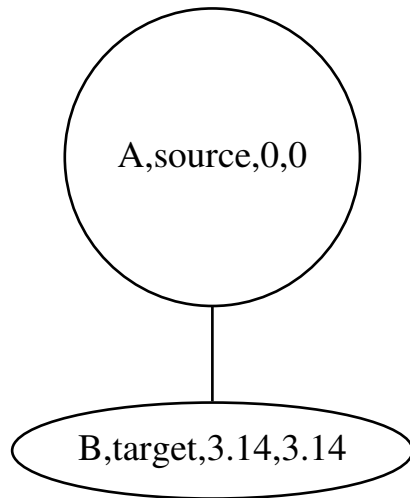


Figure 51: .svg file created from the ‘create_custom_and_selectable_vertices_k2_graph’ function (algorithm 220) its .dot file, converted from .dot file to .svg using algorithm 339

Note how the .svg changed its appearance due to the Graphviz ‘regular’ property (see chapter 25.2): the vertex labeled ‘A’ is drawn according to the Graphviz ‘regular’ attribute, which makes it a circle. The other vertex, labeled ‘B’ is not drawn as such and retained its ellipsoid appearance.

15 Working on graphs with custom and selectable vertices

This chapter shows some basic operations to do on graphs with custom and selectable vertices.

- Storing an directed/undirected graph with custom and selectable vertices as a .dot file: chapter 15.4
- Loading a directed graph with custom and selectable vertices from a .dot file: chapter 15.5
- Loading an undirected directed graph with custom and selectable vertices from a .dot file: chapter 15.6

15.1 ► Getting the vertices with a certain selectedness

15.2 ► Counting the vertices with a certain selectedness

How often is a vertex with a certain selectedness present? Here we'll find out.

Algorithm 259 Count the vertices with a certain selectedness

```
#include <string>
#include <boost/graph/properties.hpp>
#include "install_vertex_is_selected.h"

template <typename graph>
int count_vertices_with_selectedness(
    const bool selectedness,
    const graph& g
) noexcept
{
    using vd = typename graph::vertex_descriptor;

    const auto vip = vertices(g);
    const auto cnt = std::count_if(
        vip.first, vip.second,
        [g, selectedness](const vd& d)
        {
            const auto is_selected_map
                = get(boost::vertex_is_selected, g);
            return selectedness
                == get(is_selected_map, d);
        }
    );
    return static_cast<int>(cnt);
}
```

Here we use the STL `std::count_if` algorithm to count how many vertices have the desired selectedness.

Algorithm 260 shows some examples of how to do so.

Algorithm 260 Demonstration of the 'count_vertices_with_selectedness' function

```
#include <cassert>
#include "add_custom_and_selectable_vertex.h"
#include "
    create_empty_undirected_custom_and_selectable_vertices_graph
    .h"

void count_vertices_with_selectedness_demo() noexcept
{
    auto g =
        create_empty_undirected_custom_and_selectable_vertices_graph
        ();
    add_custom_and_selectable_vertex(
        my_custom_vertex("A"), true, g
    );
    add_custom_and_selectable_vertex(
        my_custom_vertex("B"), false, g
    );
    add_custom_and_selectable_vertex(
        my_custom_vertex("C"), true, g
    );
    assert(count_vertices_with_selectedness( true, g) == 2)
        ;
    assert(count_vertices_with_selectedness(false, g) == 1)
        ;
}
```

15.3 ► Adding an edge between two selected vertices

Instead of looking for an edge descriptor, one can also add an edge from two vertex descriptors. Adding an edge between two selected vertices goes as follows: use the selectedness of the vertices to get both vertex descriptors, then call 'boost::add_edge' on those two, as shown in algorithm 261.

Algorithm 261 Add an edge between two selected vertices

```
#include <cassert>
#include <string>
#include <boost/graph/adjacency_list.hpp>
#include "has_vertex_with_name.h"
#include "find_first_vertex_with_name.h"
#include "get_vertices_with_selectedness.h"
#include "count_vertices_with_selectedness.h"

template <typename graph>
typename boost::graph_traits<graph>::edge_descriptor
add_edge_between_two_selected_vertices(graph& g) noexcept
{
    assert(count_vertices_with_selectedness(true, g) == 2);
    const auto vds = get_vertices_with_selectedness(true, g);
    assert(vds.size() == 2);
    const auto aer = boost::add_edge(vds[0], vds[1], g);
    assert(aer.second);
    return aer.first;
}
```

Algorithm 262 shows how the edges can be added:

Algorithm 262 Demonstration of the 'add_edge_between_selected_vertices' function

```
#include <cassert>
#include "add_edge_between_two_selected_vertices.h"
#include "add_custom_and_selectable_vertex.h"
#include "
    create_empty_undirected_custom_and_selectable_vertices_graph
    .h"

void add_edge_between_two_selected_vertices_demo()
    noexcept
{
    auto g =
        create_empty_undirected_custom_and_selectable_vertices_graph
        ();
    add_custom_and_selectable_vertex(my_custom_vertex("Bert
        "), true, g);
    add_custom_and_selectable_vertex(my_custom_vertex("
        Ernie"), true, g);
    add_edge_between_two_selected_vertices(g);
    assert(boost::num_edges(g) == 1);
}
```

15.4 Storing a graph with custom and selectable vertices as a .dot

If you used the 'create_custom_and_selectable_vertices_k2_graph' function (algorithm 256) to produce a K_2 graph with vertices associated with (1) my_custom_vertex objects, and (2) a boolean indicating its selectedness, you can store such graphs with algorithm 263:

Algorithm 263 Storing a graph with custom and selectable vertices as a .dot file

```
#include <fstream>
#include <string>
#include <boost/graph/graphviz.hpp>
#include "install_vertex_custom_type.h"
#include "install_vertex_is_selected.h"
#include "make_custom_and_selectable_vertices_writer.h"
#include "my_custom_vertex.h"

template <typename graph>
void save_custom_and_selectable_vertices_graph_to_dot(
    const graph& g,
    const std::string& filename
)
{
    std::ofstream f(filename);
    boost::write_graphviz(f, g,
        make_custom_and_selectable_vertices_writer(
            get(boost::vertex_custom_type, g),
            get(boost::vertex_is_selected, g)
        )
    );
}
```

This code looks small, because we call the ‘make_custom_and_selectable_vertices_writer’ function, which is shown in algorithm 264:

Algorithm 264 The ‘make_custom_and_selectable_vertices_writer’ function

```
template <
    typename my_custom_vertex_map,
    typename is_selected_map
>
inline custom_and_selectable_vertices_writer<
    my_custom_vertex_map,
    is_selected_map
>
make_custom_and_selectable_vertices_writer(
    const my_custom_vertex_map& any_my_custom_vertex_map,
    const is_selected_map& any_is_selected_map
)
{
    return custom_and_selectable_vertices_writer<
        my_custom_vertex_map,
        is_selected_map
    >(
        any_my_custom_vertex_map,
        any_is_selected_map
    );
}
```

Also this function is forwarding the real work to the ‘custom_and_selectable_vertices_writer’, shown in algorithm 265:

Algorithm 265 The ‘custom_and_selectable_vertices_writer’ function

```
#include <ostream>
#include <boost/lexical_cast.hpp>
#include "is_graphviz_friendly.h"
template <
    typename my_custom_vertex_map,
    typename is_selected_map
>
class custom_and_selectable_vertices_writer {
public:
    custom_and_selectable_vertices_writer(
        my_custom_vertex_map any_my_custom_vertex_map,
        is_selected_map any_is_selected_map
    ) : m_my_custom_vertex_map{any_my_custom_vertex_map},
        m_is_selected_map{any_is_selected_map}
    {

    }

    template <class vertex_descriptor>
    void operator()(
        std::ostream& out,
        const vertex_descriptor& vd
    ) const noexcept {
        out << "[label=\""
            << get(m_my_custom_vertex_map, vd) //Can be
                Graphviz unfriendly
            << "\",_regular=\""
            << get(m_is_selected_map, vd)
            << "\"]"
        ;
    }
private:
    my_custom_vertex_map m_my_custom_vertex_map;
    is_selected_map m_is_selected_map;
};
```

Here, some interesting things are happening: the writer needs both property maps to work with (that is, the ‘my_custom_vertex’ and is_selected maps). The ‘my_custom_vertex’ are written to the Graphviz ‘label’ attribute, and the is_selected is written to the ‘regular’ attribute (see chapter 25.2 for most Graphviz attributes).

Special about this, is that even for Graphviz-unfriendly input, it still works.

15.5 Loading a directed graph with custom and selectable vertices from a .dot

When loading a graph from file, one needs to specify a type of graph. In this example, an directed graph with custom and selectable vertices is loaded, as shown in algorithm 266:

Algorithm 266 Loading a directed graph with custom vertices from a .dot file

```
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "
    create_empty_directed_custom_and_selectable_vertices_graph
    .h"
#include "is_regular_file.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex,
        boost::property<
            boost::vertex_is_selected_t, bool
        >
    >
>
>
load_directed_custom_and_selectable_vertices_graph_from_dot
(
    const std::string& dot_filename
)
{
    assert(is_regular_file(dot_filename));
    std::ifstream f(dot_filename.c_str());
    auto g =
        create_empty_directed_custom_and_selectable_vertices_graph
        ();
    boost::dynamic_properties dp(
        boost::ignore_other_properties
    );
    dp.property("label", get(boost::vertex_custom_type, g));
    ;
    dp.property("regular", get(boost::vertex_is_selected, g));
    );
    boost::read_graphviz(f, g, dp);
    return g;
}
```

In this algorithm, first it is checked if the file to load exists. Then an empty directed graph is created. Then, a `boost::dynamic_properties` is created with its default constructor, after which

- The Graphviz attribute ‘node_id’ (see chapter 25.2 for most Graphviz

attributes) is connected to a vertex its ‘my_custom_vertex’ property

- The Graphviz attribute ‘label’ is connected to a vertex its ‘my_custom_vertex’ property
- The Graphviz attribute ‘regular’ is connected to a vertex its ‘is_selected’ vertex property

Algorithm 267 shows how to use the ‘load_directed_custom_vertices_graph_from_dot’ function:

Algorithm 267 Demonstration of the ‘load_directed_custom_and_selectable_vertices_graph_from_dot’ function

```
#include <cassert>
#include "
    create_custom_and_selectable_vertices_markov_chain.h"
#include "is_regular_file.h"
#include "
    save_custom_and_selectable_vertices_graph_to_dot.h"

void
load_directed_custom_and_selectable_vertices_graph_from_dot_demo
() noexcept
{
    const auto g
        = create_custom_and_selectable_vertices_markov_chain
          ();
    const std::string filename{
        "create_custom_and_selectable_vertices_markov_chain.
        dot"
    };
    save_custom_and_selectable_vertices_graph_to_dot(
        g,
        filename
    );
    assert(is_regular_file(filename));
}
```

This demonstration shows how the Markov chain is created using the ‘create_custom_vertices_markov_chain’ function (algorithm 220), saved and then checked to exist.

15.6 Loading an undirected graph with custom and selectable vertices from a .dot

When loading a graph from file, one needs to specify a type of graph. In this example, an undirected graph with custom and selectable vertices is loaded, as shown in algorithm 268:

Algorithm 268 Loading an undirected graph with custom vertices from a .dot file

```
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "
    create_empty_undirected_custom_and_selectable_vertices_graph
    .h"
#include "install_vertex_custom_type.h"
#include "is_regular_file.h"
#include "my_custom_vertex.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex,
        boost::property<
            boost::vertex_is_selected_t, bool
        >
    >
>
>
load_undirected_custom_and_selectable_vertices_graph_from_dot
(
    const std::string& dot_filename
)
{
    assert(is_regular_file(dot_filename));
    std::ifstream f(dot_filename.c_str());
    auto g =
        create_empty_undirected_custom_and_selectable_vertices_graph
        ();
    boost::dynamic_properties dp(boost::
        ignore_other_properties);
    dp.property("label", get(boost::vertex_custom_type, g))
        ;
    dp.property("regular", get(boost::vertex_is_selected, g
        ));
    boost::read_graphviz(f, g, dp);
    return g;
}
```

The only difference with loading a directed graph, is that the initial empty graph is undirected instead. Chapter 15.5 describes the rationale of this func-

tion.

Algorithm 269 shows how to use the ‘load_undirected_custom_vertices_graph_from_dot’ function:

Algorithm 269 Demonstration of the ‘load_undirected_custom_and_selectable_vertices_graph_from_dot’ function

```
#include <cassert>
#include "create_custom_and_selectable_vertices_k2_graph.h"
#include "is_regular_file.h"
#include "
    save_custom_and_selectable_vertices_graph_to_dot.h"

void
load_undirected_custom_and_selectable_vertices_graph_from_dot_demo
() noexcept
{
    const auto g
    = create_custom_and_selectable_vertices_k2_graph();
    const std::string filename{
        "create_custom_and_selectable_vertices_k2_graph.dot"
    };
    save_custom_and_selectable_vertices_graph_to_dot(
        g,
        filename
    );
    assert(is_regular_file(filename));
}
```

This demonstration shows how K_2 with custom vertices is created using the ‘create_custom_vertices_k2_graph’ function (algorithm 223), saved and then checked to exist.

16 Building graphs with custom edges and vertices

Up until now, the graphs created have had edges and vertices with the built-in name property. In this chapter, graphs will be created, in which the edges and vertices can have a custom ‘my_custom_edge’ and ‘my_custom_edge’ type¹².

- An empty directed graph that allows for custom edges and vertices: see chapter 16.3

¹²I do not intend to be original in naming my data types

- An empty undirected graph that allows for custom edges and vertices: see chapter 16.4
- A two-state Markov chain with custom edges and vertices: see chapter 16.7
- K_3 with custom edges and vertices: see chapter 16.8

In the process, some basic (sometimes bordering trivial) functions are shown:

- Creating the custom edge class: see chapter 16.1
- Installing the new edge property: see chapter 16.2
- Adding a custom edge: see chapter 16.5

These functions are mostly there for completion and showing which data types are used.

16.1 Creating the custom edge class

In this example, I create a custom edge class. Here I will show the header file of it, as the implementation of it is not important yet.

Algorithm 270 Declaration of `my_custom_edge`

```
#include <string>
#include <iosfwd>

class my_custom_edge
{
public:
    explicit my_custom_edge(
        const std::string& name = "",
        const std::string& description = "",
        const double width = 1.0,
        const double height = 1.0
    ) noexcept;
    const std::string& get_description() const noexcept;
    const std::string& get_name() const noexcept;
    double get_width() const noexcept;
    double get_height() const noexcept;
private:
    std::string m_name;
    std::string m_description;
    double m_width;
    double m_height;
};

bool operator==(const my_custom_edge& lhs, const
    my_custom_edge& rhs) noexcept;
bool operator!=(const my_custom_edge& lhs, const
    my_custom_edge& rhs) noexcept;
std::ostream& operator<<(std::ostream& os, const
    my_custom_edge& v) noexcept;
std::istream& operator>>(std::istream& os, my_custom_edge
    & v) noexcept;
```

`my_custom_edge` is a class that has multiple properties: two doubles ‘`m_width`’ (‘`m_`’ stands for member) and ‘`m_height`’, and two `std::string`s `m_name` and `m_description`. ‘`my_custom_edge`’ is copyable, but cannot trivially be converted to a ‘`std::string`.’ ‘`my_custom_edge`’ is comparable for equality (that is, `operator==` is defined).

Special characters like comma’s, quotes and whitespace cannot be streamed without problems. The function ‘`graphviz_encode`’ (algorithm 335) can convert the elements to be streamed to a Graphviz-friendly version, which can be decoded by ‘`graphviz_decode`’ (algorithm 336).

16.2 Installing the new edge property

Installing a new property would have been easier, if ‘more C++ compilers were standards conformant’ ([8], chapter 3.6, footnote at page 52). Boost.Graph uses the BOOST_INSTALL_PROPERTY macro to allow using a custom property:

Algorithm 271 Installing the edge_custom_type property

```
#include <boost/graph/properties.hpp>

namespace boost {
    enum edge_custom_type_t { edge_custom_type = 3142 };
    BOOST_INSTALL_PROPERTY(edge, custom_type);
}
```

The enum value 3142 must be unique.

16.3 Create an empty directed graph with custom edges and vertices

Algorithm 272 Creating an empty directed graph with custom edges and vertices

```
#include <boost/graph/adjacency_list.hpp>
#include "install_edge_custom_type.h"
#include "install_vertex_custom_type.h"
#include "my_custom_edge.h"
#include "my_custom_vertex.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex
    >,
    boost::property<
        boost::edge_custom_type_t, my_custom_edge
    >
>
create_empty_directed_custom_edges_and_vertices_graph()
    noexcept
{
    return {};
}
```

This code is very similar to the code described in chapter 12.3, except that there is a new, fifth template argument:

```
boost::property<boost::edge_custom_type_t, my_edge>
```

This can be read as: “edges have the property ‘boost::edge_custom_type_t’, which is of data type ‘my_custom_edge’”. Or simply: “edges have a custom type called my_custom_edge”.

Demo:

Algorithm 273 Demonstration of the ‘create_empty_directed_custom_edges_and_vertices_graph’ function

```
#include "
    create_empty_directed_custom_edges_and_vertices_graph.h"

void
    create_empty_directed_custom_edges_and_vertices_graph_demo
    () noexcept
{
    const auto g =
        create_empty_directed_custom_edges_and_vertices_graph
        ();
    assert(boost::num_edges(g) == 0);
    assert(boost::num_vertices(g) == 0);
}
```

16.4 Create an empty undirected graph with custom edges and vertices

Algorithm 274 Creating an empty undirected graph with custom edges and vertices

```
#include <boost/graph/adjacency_list.hpp>
#include "install_edge_custom_type.h"
#include "install_vertex_custom_type.h"
#include "my_custom_vertex.h"
#include "my_custom_edge.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex
    >,
    boost::property<
        boost::edge_custom_type_t, my_custom_edge
    >
>
create_empty_undirected_custom_edges_and_vertices_graph()
    noexcept
{
    return {};
}
```

This code is very similar to the code described in chapter 16.3, except that the directedness (the third template argument) is undirected (due to the `boost::undirectedS`).

Demo:

Algorithm 275 Demonstration of the ‘create_empty_undirected_custom_edges_and_vertices_graph’ function

```
#include <cassert>
#include "
    create_empty_undirected_custom_edges_and_vertices_graph
    .h"

void
    create_empty_undirected_custom_edges_and_vertices_graph_demo
    () noexcept
{
    const auto g
        =
            create_empty_undirected_custom_edges_and_vertices_graph
            ();
    assert(boost::num_edges(g) == 0);
    assert(boost::num_vertices(g) == 0);
}
```

16.5 Add a custom edge

Adding a custom edge is very similar to adding a named edge (chapter 6.3).

Algorithm 276 Add a custom edge

```
#include <cassert>
#include <boost/graph/adjacency_list.hpp>
#include "install_edge_custom_type.h"
#include "my_custom_edge.h"

template <typename graph>
typename boost::graph_traits<graph>::edge_descriptor
add_custom_edge(
    const my_custom_edge& v,
    graph& g
) noexcept
{
    static_assert(!std::is_const<graph>::value, "graph_
        cannot_be_const");

    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);

    const auto aer
        = boost::add_edge(vd_a, vd_b, g);
    assert(aer.second);
    const auto my_edge_map
        = get(boost::edge_custom_type, g);
    put(my_edge_map, aer.first, v);
    return aer.first;
}
```

When having added a new (abstract) edge to the graph, the edge descriptor is used to set the my_edge in the graph its my_custom_edge map (using 'get(boost::edge_custom_type,g)').

Here is the demo:

Algorithm 277 Demo of ‘add_custom_edge’

```
#include <cassert>
#include "add_custom_edge.h"
#include "
    create_empty_directed_custom_edges_and_vertices_graph.
    h"
#include "
    create_empty_undirected_custom_edges_and_vertices_graph
    .h"

void add_custom_edge_demo() noexcept
{
    auto g =
        create_empty_directed_custom_edges_and_vertices_graph
        ();
    add_custom_edge(my_custom_edge("X"), g);
    assert(boost::num_vertices(g) == 2);
    assert(boost::num_edges(g) == 1);

    auto h =
        create_empty_undirected_custom_edges_and_vertices_graph
        ();
    add_custom_edge(my_custom_edge("Y"), h);
    assert(boost::num_vertices(h) == 2);
    assert(boost::num_edges(h) == 1);
}
```

16.6 Getting the custom edges my_edges

When the edges of a graph have an associated ‘my_custom_edge’, one can extract these all as such:

Algorithm 278 Get the edges' my_custom_edges

```
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include "install_edge_custom_type.h"
#include "my_custom_edge.h"

template <typename graph>
std::vector<my_custom_edge> get_my_custom_edges(
    const graph& g
) noexcept
{
    using ed = typename boost::graph_traits<graph>::
        edge_descriptor;
    std::vector<my_custom_edge> v(boost::num_edges(g));
    const auto eip = edges(g);
    std::transform(eip.first, eip.second, std::begin(v),
        [g](const ed d)
        {
            const auto my_custom_edges_map
                = get(boost::edge_custom_type, g);
            return get(my_custom_edges_map, d);
        }
    );
    return v;
}
```

The 'my_custom_edge' object associated with the edges are obtained from a boost::property_map and then put into a std::vector.

Note: the order of the my_custom_edge objects may be different after saving and loading.

When trying to get the edges' my_custom_edge objects from a graph without custom edges objects associated, you will get the error 'formed reference to void' (see chapter 24.1).

16.7 Creating a Markov-chain with custom edges and vertices

16.7.1 Graph

Figure 52 shows the graph that will be reproduced:

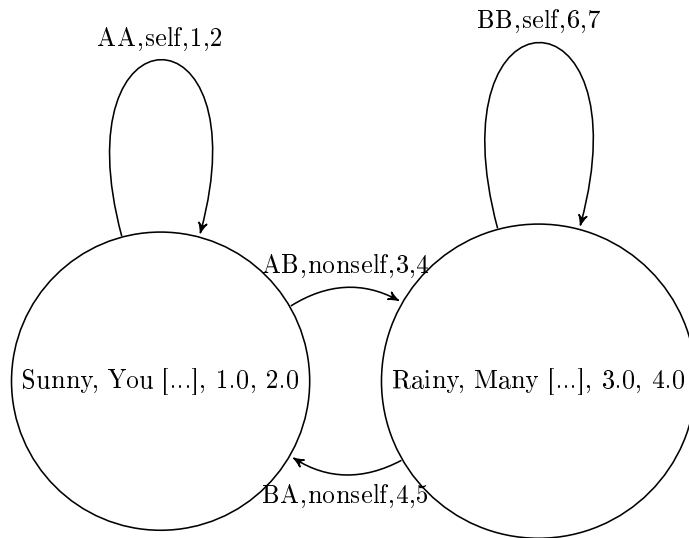


Figure 52: A two-state Markov chain where the edges and vertices have custom properies. The edges' and vertices' properties are nonsensical

16.7.2 Function to create such a graph

Here is the code creating a two-state Markov chain with custom edges and vertices:

Algorithm 279 Creating the two-state Markov chain as depicted in figure 52

```
#include <cassert>
#include "
    create_empty_directed_custom_edges_and_vertices_graph.
    h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex
    >,
    boost::property<
        boost::edge_custom_type_t, my_custom_edge
    >
>
>
create_custom_edges_and_vertices_markov_chain() noexcept
{
    auto g
        =
            create_empty_directed_custom_edges_and_vertices_graph
                ();
    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);
    const auto aer_aa = boost::add_edge(vd_a, vd_a, g);
    assert(aer_aa.second);
    const auto aer_ab = boost::add_edge(vd_a, vd_b, g);
    assert(aer_ab.second);
    const auto aer_ba = boost::add_edge(vd_b, vd_a, g);
    assert(aer_ba.second);
    const auto aer_bb = boost::add_edge(vd_b, vd_b, g);
    assert(aer_bb.second);

    auto my_custom_vertexes_map = get(
        boost::vertex_custom_type, g
    );
    put(my_custom_vertexes_map, vd_a,
        my_custom_vertex("Sunny", "Yellow_thing", 1.0, 2.0)
    );
    put(my_custom_vertexes_map, vd_b,
        my_custom_vertex("Rainy", "Grey_things", 3.0, 4.0)
    );

    auto my_edges_map = get(
        boost::edge_custom_type, g
    );
    put(my_edges_map, aer_aa.first,
        my_custom_edge("Sometimes", "20%", 1.0, 2.0)
    );
    put(my_edges_map, aer_ab.first,
        my_custom_edge("Often", "80%", 3.0, 4.0)
    );
    put(my_edges_map, aer_ba.first,
        my_custom_edge("Rarely", "10%", 5.0, 6.0)
    );
}
```

16.7.3 Creating such a graph

Here is the demo:

Algorithm 280 Demo of the ‘create_custom_edges_and_vertices_markov_chain’ function (algorithm 279)

```
#include <cassert>
#include "create_custom_edges_and_vertices_markov_chain.h"
"
#include "get_my_custom_vertexes.h"
#include "install_vertex_custom_type.h"
#include "my_custom_vertex.h"

void create_custom_edges_and_vertices_markov_chain_demo()
    noexcept
{
    const auto g
        = create_custom_edges_and_vertices_markov_chain();
    const std::vector<my_custom_vertex>
        expected_my_custom_vertexes{
            my_custom_vertex("Sunny",
                "Yellow_thing",1.0,2.0
            ),
            my_custom_vertex("Rainy",
                "Grey_things",3.0,4.0
            )
        };
    const std::vector<my_custom_vertex>
        vertex_my_custom_vertexes{
            get_my_custom_vertexes(g)
        };
    assert(expected_my_custom_vertexes
        == vertex_my_custom_vertexes
    );
}
```

16.7.4 The .dot file produced

Algorithm 281 .dot file created from the ‘create_custom_edges_and_vertices_markov_chain’ function (algorithm 279), converted from graph to .dot file using algorithm 48

```

digraph G {
0[label="Sunny,Yellow$$$$SPACE$$$thing,1,1"];
1[label="Rainy,Grey$$$$SPACE$$$things,3,3"];
0->0 [label="Sometimes,20%,1,2"];
0->1 [label="Often,80%,3,4"];
1->0 [label="Rarely,10%,5,6"];
1->1 [label="Mostly,90%,7,8"];
}

```

16.7.5 The .svg file produced

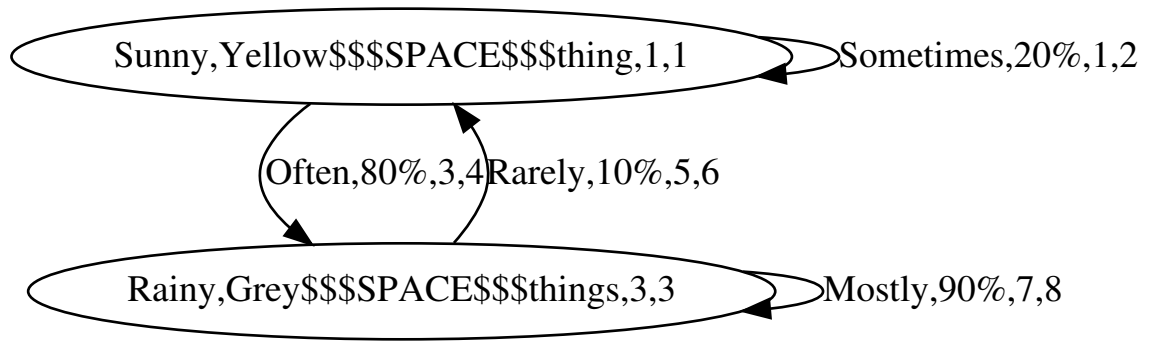


Figure 53: .svg file created from the ‘create_custom_edges_and_vertices_markov_chain’ function (algorithm 220) its .dot file, converted from .dot file to .svg using algorithm 339

16.8 Creating K_3 with custom edges and vertices

Instead of using edges with a name, or other properties, here we use a custom edge class called ‘my_custom_edge’.

16.8.1 Graph

We reproduce the K_3 with named edges and vertices of chapter 6.8 , but with our custom edges and vertices instead:

[graph here]

16.8.2 Function to create such a graph

Algorithm 282 Creating K_3 as depicted in figure 30

```
#include "
    create_empty_undirected_custom_edges_and_vertices_graph
    .h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex
    >,
    boost::property<
        boost::edge_custom_type_t, my_custom_edge
    >
>
create_custom_edges_and_vertices_k3_graph() noexcept
{
    auto g
        =
            create_empty_undirected_custom_edges_and_vertices_graph
            ();
    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);
    const auto vd_c = boost::add_vertex(g);
    const auto aer_a = boost::add_edge(vd_a, vd_b, g);
    const auto aer_b = boost::add_edge(vd_b, vd_c, g);
    const auto aer_c = boost::add_edge(vd_c, vd_a, g);
    assert(aer_a.second);
    assert(aer_b.second);
    assert(aer_c.second);

    auto my_custom_vertex_map = get(
        boost::vertex_custom_type, g
    );
    put(my_custom_vertex_map, vd_a,
        my_custom_vertex("top", "source", 0.0, 0.0)
    );
    put(my_custom_vertex_map, vd_b,
        my_custom_vertex("right", "target", 3.14, 0)
    );
    put(my_custom_vertex_map, vd_c,
        my_custom_vertex("left", "target", 0, 3.14)
    );

    auto my_edge_map = get(boost::edge_custom_type, g);
    put(my_edge_map, aer_a.first,
        my_custom_edge("AB", "first", 0.0, 0.0)
    );
    put(my_edge_map, aer_b.first,
        my_custom_edge("BC", "second", 3.14, 3.14)
    );
    put(my_edge_map, aer_c.first,
```


Most of the code is a slight modification of algorithm 128. In the end, the `my_edges` and `my_vertices` are obtained as a `boost::property_map` and set with the `'my_custom_edge'` and `'my_custom_vertex'` objects.

16.8.3 Creating such a graph

Here is the demo:

Algorithm 283 Demo of the `'create_custom_edges_and_vertices_k3_graph'` function (algorithm 282)

```
#include <cassert>
#include "add_custom_edge.h"
#include "add_custom_vertex.h"
#include "create_custom_edges_and_vertices_k3_graph.h"

void create_custom_edges_and_vertices_k3_graph_demo()
    noexcept
{
    auto g
        = create_custom_edges_and_vertices_k3_graph();
    assert(boost::num_edges(g) == 3);
    assert(boost::num_vertices(g) == 3);
    add_custom_vertex(my_custom_vertex("v"), g);
    add_custom_edge(my_custom_edge("e"), g);
}
```

16.8.4 The .dot file produced

Algorithm 284 .dot file created from the `'create_custom_edges_and_vertices_markov_chain'` function (algorithm 282), converted from graph to .dot file using algorithm 48

```
graph G {
0[label="top,source,0,0"];
1[label="right,target,3.14,3.14"];
2[label="left,target,0,0"];
0--1 [label="AB,first,0,0"];
1--2 [label="BC,second,3.14,3.14"];
2--0 [label="CA,third,3.14,3.14"];
}
```

16.8.5 The .svg file produced

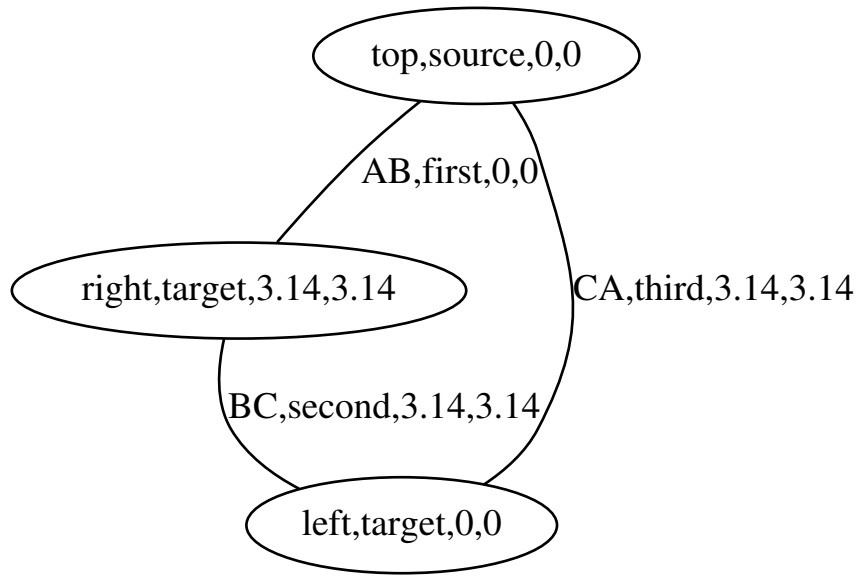


Figure 54: .svg file created from the ‘create_custom_edges_and_vertices_k3_graph’ function (algorithm 220) its .dot file, converted from .dot file to .svg using algorithm 339

17 Working on graphs with custom edges and vertices

17.1 Has a my_custom_edge

Before modifying our edges, let’s first determine if we can find an edge by its custom type (‘my_custom_edge’) in a graph. After obtaining a my_custom_edge map, we obtain the edge iterators, dereference these to obtain the edge descriptors and then compare each edge its my_custom_edge with the one desired.

Algorithm 285 Find if there is a custom edge with a certain `my_custom_edge`

```
#include <boost/graph/properties.hpp>
#include "install_edge_custom_type.h"
#include "my_custom_edge.h"

template <typename graph>
bool has_custom_edge_with_my_edge(
    const my_custom_edge& e,
    const graph& g
) noexcept
{
    using ed = typename boost::graph_traits<graph>::
        edge_descriptor;
    const auto eip = edges(g);
    return std::find_if(eip.first, eip.second,
        [e, g](const ed& d)
        {
            const auto my_edges_map
                = get(boost::edge_custom_type, g);
            return get(my_edges_map, d) == e;
        }
    ) != eip.second;
}
```

This function can be demonstrated as in algorithm 286, where a certain ‘`my_custom_edge`’ cannot be found in an empty graph. After adding the desired `my_custom_edge`, it is found.

Algorithm 286 Demonstration of the ‘has_custom_edge_with_my_edge’ function

```
#include <cassert>
#include "add_custom_edge.h"
#include "
    create_empty_undirected_custom_edges_and_vertices_graph
    .h"
#include "has_custom_edge_with_my_edge.h"

void has_custom_edge_with_my_edge_demo() noexcept
{
    auto g
        =
            create_empty_undirected_custom_edges_and_vertices_graph
                ();
    assert (
        !has_custom_edge_with_my_edge(
            my_custom_edge("Edward"), g
        )
    );
    add_custom_edge(my_custom_edge("Edward"), g);
    assert (
        has_custom_edge_with_my_edge(
            my_custom_edge("Edward"), g
        )
    );
}
```

Note that this function only finds if there is at least one edge with that my_custom_edge: it does not tell how many edges with that my_custom_edge exist in the graph.

17.2 Find a my_custom_edge

Where STL functions work with iterators, here we obtain an edge descriptor (see chapter 2.12) to obtain a handle to the desired edge. Algorithm 287 shows how to obtain an edge descriptor to the first edge found with a specific my_custom_edge value.

Algorithm 287 Find the first custom edge with a certain `my_custom_edge`

```
#include <cassert>
#include <boost/graph/graph_traits.hpp>
#include "has_custom_edge_with_my_edge.h"
#include "install_edge_custom_type.h"
#include "my_custom_edge.h"

template <typename graph>
typename boost::graph_traits<graph>::edge_descriptor
find_first_custom_edge_with_my_edge(
    const my_custom_edge& e,
    const graph& g
) noexcept
{
    using ed = typename boost::graph_traits<graph>::
        edge_descriptor;
    const auto eip = edges(g);
    const auto i = std::find_if(
        eip.first, eip.second,
        [e,g](const ed d) {
            const auto my_edges_map = get(boost::
                edge_custom_type, g);
            return get(my_edges_map, d) == e;
        }
    );
    assert(i != eip.second);
    return *i;
}
```

With the edge descriptor obtained, one can read and modify the edge and the vertices surrounding it. Algorithm 288 shows some examples of how to do so.

Algorithm 288 Demonstration of the ‘find_first_custom_edge_with_my_edge’ function

```
#include <cassert>

#include "create_custom_edges_and_vertices_k3_graph.h"
#include "find_first_custom_edge_with_my_edge.h"

void find_first_custom_edge_with_my_edge_demo() noexcept
{
    const auto g
        = create_custom_edges_and_vertices_k3_graph();
    const auto ed
        = find_first_custom_edge_with_my_edge(
            my_custom_edge("AB", "first", 0.0, 0.0),
            g
        );
    assert(boost::source(ed, g)
        != boost::target(ed, g)
    );
}
```

17.3 Get an edge its my_custom_edge

To obtain the my_edge from an edge descriptor, one needs to pull out the my_custom_edges map and then look up the my_edge of interest.

Algorithm 289 Get a vertex its my_custom_vertex from its vertex descriptor

```
#include <boost/graph/graph_traits.hpp>
#include "install_edge_custom_type.h"
#include "my_custom_edge.h"

template <typename graph>
my_custom_edge get_my_custom_edge(
    const typename boost::graph_traits<graph>::
        edge_descriptor& vd,
    const graph& g
) noexcept
{
    const auto my_edge_map
        = get(boost::edge_custom_type, g);
    return get(my_edge_map, vd);
}
```

To use ‘get_custom_edge_my_custom_edge’, one first needs to obtain an edge descriptor. Algorithm 290 shows a simple example.

Algorithm 290 Demonstration if the ‘get_custom_edge_my_edge’ function

```
#include <cassert>

#include "add_custom_edge.h"
#include "
    create_empty_undirected_custom_edges_and_vertices_graph
    .h"
#include "find_first_custom_edge_with_my_edge.h"
#include "get_my_custom_edge.h"

void get_my_custom_edge_demo() noexcept
{
    auto g
        =
            create_empty_undirected_custom_edges_and_vertices_graph
            ();
    const my_custom_edge edge{"Dex"};
    add_custom_edge(edge, g);
    const auto ed
        = find_first_custom_edge_with_my_edge(edge, g);
    assert(get_my_custom_edge(ed, g) == edge);
}
```

17.4 Set an edge its my_custom_edge

If you know how to get the my_custom_edge from an edge descriptor, setting it is just as easy, as shown in algorithm 291.

Algorithm 291 Set a custom edge its `my_custom_edge` from its edge descriptor

```
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "install_edge_custom_type.h"
#include "my_custom_edge.h"

template <typename graph>
void set_my_custom_edge(
    const my_custom_edge& name,
    const typename boost::graph_traits<graph>::
        edge_descriptor& vd,
    graph& g
) noexcept
{
    static_assert(!std::is_const<graph>::value, "graph_
        cannot_be_const");

    auto my_edge_map = get(boost::edge_custom_type, g);
    put(my_edge_map, vd, name);
}
```

To use ‘`set_my_custom_edge`’, one first needs to obtain an edge descriptor. Algorithm 292 shows a simple example.

Algorithm 292 Demonstration if the ‘set_my_custom_edge’ function

```
#include <cassert>

#include "add_custom_edge.h"
#include "
    create_empty_undirected_custom_edges_and_vertices_graph
    .h"
#include "find_first_custom_edge_with_my_edge.h"
#include "get_my_custom_edge.h"
#include "set_my_custom_edge.h"

void set_my_custom_edge_demo() noexcept
{
    auto g
        =
            create_empty_undirected_custom_edges_and_vertices_graph
            ();
    const my_custom_edge old_edge{"Dex"};
    add_custom_edge(old_edge, g);
    const auto vd
        = find_first_custom_edge_with_my_edge(old_edge, g);
    assert(get_my_custom_edge(vd, g)
        == old_edge
    );
    const my_custom_edge new_edge{"Diggy"};
    set_my_custom_edge(new_edge, vd, g);
    assert(get_my_custom_edge(vd, g)
        == new_edge
    );
}
```

17.5 ► Counting the edges with a certain selectedness

How often is an edge with a certain selectedness present? Here we'll find out.

Algorithm 293 Count the edges with a certain selectedness

```
#include <string>
#include <boost/graph/properties.hpp>
#include "install_edge_is_selected.h"

template <typename graph>
int count_edges_with_selectedness(
    const bool selectedness,
    const graph& g
) noexcept
{
    using ed = typename graph::edge_descriptor;

    const auto eip = edges(g);
    const auto cnt = std::count_if(
        eip.first, eip.second,
        [g, selectedness](const ed& d)
        {
            const auto is_selected_map
                = get(boost::edge_is_selected, g);
            return selectedness
                == get(is_selected_map, d);
        }
    );
    return static_cast<int>(cnt);
}
```

Here we use the STL `std::count_if` algorithm to count how many vertices have the desired selectedness.

Algorithm 294 shows some examples of how to do so.

Algorithm 294 Demonstration of the 'count_edges_with_selectedness' function

```
#include <cassert>
#include "count_edges_with_selectedness.h"
#include "
    create_empty_directed_custom_and_selectable_edges_and_vertices_graph
    .h"
#include "add_custom_and_selectable_edge.h"

void count_edges_with_selectedness_demo() noexcept
{
    auto g =
        create_empty_directed_custom_and_selectable_edges_and_vertices_graph
        ();
    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);
    add_custom_and_selectable_edge(
        vd_a, vd_b, my_custom_edge("AB"), true, g
    );
    add_custom_and_selectable_edge(
        vd_a, vd_a, my_custom_edge("AA"), false, g
    );
    assert(count_edges_with_selectedness(true, g) == 1);
    assert(count_edges_with_selectedness(false, g) == 1);
}
```

17.6 Storing a graph with custom edges and vertices as a .dot

If you used the `create_custom_edges_and_vertices_k3_graph` function (algorithm 282) to produce a K_3 graph with edges and vertices associated with `my_custom_edge` and `my_custom_vertex` objects, you can store these `my_custom_edges` and `my_custom_vertex-es` additionally with algorithm 295:

Algorithm 295 Storing a graph with custom edges and vertices as a .dot file

```
#include <fstream>
#include <string>
#include <boost/graph/graphviz.hpp>
#include <boost/graph/properties.hpp>
#include "get_my_custom_edge.h"
#include "get_my_custom_vertex.h"
#include "my_custom_vertex.h"

template <typename graph>
void save_custom_edges_and_vertices_graph_to_dot(
    const graph& g,
    const std::string& filename
)
{
    using vd = typename graph::vertex_descriptor;
    using ed = typename graph::edge_descriptor;

    std::ofstream f(filename);
    boost::write_graphviz(
        f,
        g,
        [g](
            std::ostream& out, const vd& d) {
                const my_custom_vertex m{
                    get_my_custom_vertex(d, g)
                };
                out << "[label=\"" << m << "\"]";
            },
        [g](std::ostream& out, const ed& d) {
                const my_custom_edge& m{
                    get_my_custom_edge(d, g)
                };
                out << "[label=\"" << m << "\"]";
            }
        );
}
```

17.7 Load a directed graph with custom edges and vertices from a .dot file

When loading a graph from file, one needs to specify a type of graph. In this example, an directed graph with custom edges and vertices is loaded, as shown in algorithm 296:

Algorithm 296 Loading a directed graph with custom edges and vertices from a .dot file

```

#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "
    create_empty_directed_custom_edges_and_vertices_graph.
    h"
#include "is_regular_file.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex
    >,
    boost::property<
        boost::edge_custom_type_t, my_custom_edge
    >
>
>
load_directed_custom_edges_and_vertices_graph_from_dot(
    const std::string& dot_filename
)
{
    assert(is_regular_file(dot_filename));
    std::ifstream f(dot_filename.c_str());
    auto g =
        create_empty_directed_custom_edges_and_vertices_graph
        ();
    boost::dynamic_properties dp(boost::
        ignore_other_properties);
    dp.property("label", get(boost::vertex_custom_type, g));
    ;
    dp.property("edge_id", get(boost::edge_custom_type, g));
    ;
    dp.property("label", get(boost::edge_custom_type, g));
    boost::read_graphviz(f, g, dp);
    return g;
}

```

In this algorithm, first it is checked if the file to load exists. Then an empty directed graph is created. Next to this, a `boost::dynamic_properties` is created with its default constructor, after which we direct the `boost::dynamic_properties` to find a 'node_id' and 'label' in the vertex name map, 'edge_id' and 'label' to

the edge name map. From this and the empty graph, ‘boost::read_graphviz’ is called to build up the graph.

Algorithm 297 shows how to use the ‘load_directed_custom_edges_and_vertices_graph_from_dot’ function:

Algorithm 297 Demonstration of the ‘load_directed_custom_edges_and_vertices_graph_from_dot’ function

```
#include "create_custom_edges_and_vertices_markov_chain.h"
"
#include "get_my_custom_vertexes.h"
#include "
    load_directed_custom_edges_and_vertices_graph_from_dot
    .h"
#include "save_custom_edges_and_vertices_graph_to_dot.h"

void
load_directed_custom_edges_and_vertices_graph_from_dot_demo
() noexcept
{
    using boost::num_edges;
    using boost::num_vertices;

    const auto g
        = create_custom_edges_and_vertices_markov_chain();
    const std::string filename{
        "create_custom_edges_and_vertices_markov_chain.dot"
    };
    save_custom_edges_and_vertices_graph_to_dot(g, filename
    );
    const auto h
        =
            load_directed_custom_edges_and_vertices_graph_from_dot
            (
                filename
            );
    assert(num_edges(g) == num_edges(h));
    assert(num_vertices(g) == num_vertices(h));
    assert(get_my_custom_vertexes(g)
        == get_my_custom_vertexes(h)
    );
}
```

This demonstration shows how the Markov chain is created using the ‘create_custom_edges_and_vertices_markov_chain’ function (algorithm 279), saved

and then loaded.

17.8 Load an undirected graph with custom edges and vertices from a .dot file

When loading a graph from file, one needs to specify a type of graph. In this example, an undirected graph with custom edges and vertices is loaded, as shown in algorithm 298:

Algorithm 298 Loading an undirected graph with custom edges and vertices from a .dot file

```
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "
    create_empty_undirected_custom_edges_and_vertices_graph
    .h"
#include "is_regular_file.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex
    >,
    boost::property<
        boost::edge_custom_type_t, my_custom_edge
    >
>
load_undirected_custom_edges_and_vertices_graph_from_dot(
    const std::string& dot_filename
)
{
    assert(is_regular_file(dot_filename));
    std::ifstream f(dot_filename.c_str());
    auto g =
        create_empty_undirected_custom_edges_and_vertices_graph
        ();
    boost::dynamic_properties dp(boost::
        ignore_other_properties);
    dp.property("label", get(boost::vertex_custom_type, g));
    ;
    dp.property("edge_id", get(boost::edge_custom_type, g));
    ;
    dp.property("label", get(boost::edge_custom_type, g));
    boost::read_graphviz(f, g, dp);
    return g;
}
```

The only difference with loading a directed graph, is that the initial empty graph is undirected instead. Chapter 17.7 describes the rationale of this function.

Algorithm 299 shows how to use the ‘load_undirected_custom_vertices_graph_from_dot’

function:

Algorithm 299 Demonstration of the ‘load_undirected_custom_edges_and_vertices_graph_from_dot’ function

```
#include "create_custom_edges_and_vertices_k3_graph.h"
#include "
    load_undirected_custom_edges_and_vertices_graph_from_dot
    .h"
#include "save_custom_edges_and_vertices_graph_to_dot.h"
#include "get_my_custom_vertexes.h"

void
load_undirected_custom_edges_and_vertices_graph_from_dot_demo
() noexcept
{
    using boost::num_edges;
    using boost::num_vertices;

    const auto g
        = create_custom_edges_and_vertices_k3_graph();
    const std::string filename{
        "create_custom_edges_and_vertices_k3_graph.dot"
    };
    save_custom_edges_and_vertices_graph_to_dot(g, filename
    );
    const auto h
        =
            load_undirected_custom_edges_and_vertices_graph_from_dot
            (filename);
    assert(num_edges(g) == num_edges(h));
    assert(num_vertices(g) == num_vertices(h));
    assert(get_my_custom_vertexes(g) ==
        get_my_custom_vertexes(h));
}
```

This demonstration shows how K_2 with custom vertices is created using the ‘create_custom_vertices_k2_graph’ function (algorithm 223), saved and then loaded. The loaded graph is checked to be a graph similar to the original.

18 Building graphs with custom and selectable edges and vertices

Now also the edge can be selected

- An empty directed graph that allows for custom and selectable vertices: see chapter 18.2
- An empty undirected graph that allows for custom and selectable vertices: see chapter 18.3
- A two-state Markov chain with custom and selectable vertices: see chapter 18.5
- K_3 with custom and selectable vertices: see chapter 18.6

In the process, some basic (sometimes bordering trivial) functions are shown:

- Installing the new edge property: see chapter 18.1
- Adding a custom and selectable vertex: see chapter 18.4

These functions are mostly there for completion and showing which data types are used.

18.1 Installing the new `is_selected` property

Installing a new property would have been easier, if ‘more C++ compilers were standards conformant’ ([8], chapter 3.6, footnote at page 52). Boost.Graph uses the `BOOST_INSTALL_PROPERTY` macro to allow using a custom property:

Algorithm 300 Installing the `edge_is_selected` property

```
#include <boost/graph/properties.hpp>

namespace boost {
    enum edge_is_selected_t { edge_is_selected = 314159 };
    BOOST_INSTALL_PROPERTY(edge, is_selected);
}
```

The enum value 31415 must be unique.

18.2 Create an empty directed graph with custom and selectable edges and vertices

Algorithm 301 Creating an empty directed graph with custom and selectable edges and vertices

```
#include <boost/graph/adjacency_list.hpp>
#include "install_edge_custom_type.h"
#include "install_edge_is_selected.h"
#include "install_vertex_custom_type.h"
#include "install_vertex_is_selected.h"
#include "my_custom_edge.h"
#include "my_custom_vertex.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex,
        boost::property<
            boost::vertex_is_selected_t, bool
        >
    >,
    >,
    boost::property<
        boost::edge_custom_type_t, my_custom_edge,
        boost::property<
            boost::edge_is_selected_t, bool
        >
    >
>
create_empty_directed_custom_and_selectable_edges_and_vertices_graph
    () noexcept
{
    return {};
}
```

This code is very similar to the code described in chapter 12.3, except that there is a new, fifth template argument:

```
boost::property<boost::edge_custom_type_t, my_custom_edge,
    boost::property<boost::edge_is_selected_t, bool,
>
```

This can be read as: “edges have two properties: an associated custom type (of type `my_custom_edge`) and an associated `is_selected` property (of type `bool`)”.

Demo:

Algorithm 302 Demonstration of the ‘create_empty_directed_custom_and_selectable_edges_and_vertices_graph’ function

```
#include "
    create_empty_directed_custom_and_selectable_edges_and_vertices_graph
    .h"

void
    create_empty_directed_custom_and_selectable_edges_and_vertices_graph_demo
    () noexcept
{
    const auto g
        =
            create_empty_directed_custom_and_selectable_edges_and_vertices_graph
            ();
    assert(boost::num_edges(g) == 0);
    assert(boost::num_vertices(g) == 0);
}
```

18.3 Create an empty undirected graph with custom and selectable edges and vertices

Algorithm 303 Creating an empty undirected graph with custom and selectable edges and vertices

```
#include <boost/graph/adjacency_list.hpp>
#include "install_edge_custom_type.h"
#include "install_edge_is_selected.h"
#include "install_vertex_custom_type.h"
#include "install_vertex_is_selected.h"
#include "my_custom_edge.h"
#include "my_custom_vertex.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex,
        boost::property<
            boost::vertex_is_selected_t, bool
        >
    >,
    >,
    boost::property<
        boost::edge_custom_type_t, my_custom_edge,
        boost::property<
            boost::edge_is_selected_t, bool
        >
    >
>
create_empty_undirected_custom_and_selectable_edges_and_vertices_graph
() noexcept
{
    return {};
}
```

This code is very similar to the code described in chapter 18.2, except that the directedness (the third template argument) is undirected (due to the `boost::undirectedS`).

Demo:

Algorithm 304 Demonstration of the ‘create_empty_undirected_custom_and_selectable_edges_and_vertices_graph’ function

```
#include "
    create_empty_undirected_custom_and_selectable_edges_and_vertices_graph
    .h"

void
    create_empty_undirected_custom_and_selectable_edges_and_vertices_graph_demo
    () noexcept
{
    const auto g
        =
            create_empty_undirected_custom_and_selectable_edges_and_vertices_graph
            ();
    assert(boost::num_edges(g) == 0);
    assert(boost::num_vertices(g) == 0);
}
```

18.4 Add a custom and selectable edge

Adding a custom and selectable edge is very similar to adding a custom and selectable vertex (chapter 14.4).

Algorithm 305 Add a custom and selectable edge

```
#include <type_traits>
#include <boost/graph/adjacency_list.hpp>
#include "install_edge_custom_type.h"
#include "install_edge_is_selected.h"
#include "my_custom_edge.h"

template <typename graph>
std::pair<
    typename boost::graph_traits<graph>::edge_descriptor,
    bool
>
add_custom_and_selectable_edge(
    const typename boost::graph_traits<graph>::
        vertex_descriptor& vd_from,
    const typename boost::graph_traits<graph>::
        vertex_descriptor& vd_to,
    const my_custom_edge& edge,
    const bool is_selected,
    graph& g
) noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const");
};

const auto aer = boost::add_edge(vd_from, vd_to, g);
assert(aer.second);

const auto my_custom_edge_map
    = get(boost::edge_custom_type,
        g
    );
put(my_custom_edge_map, aer.first, edge);

const auto is_selected_map
    = get(boost::edge_is_selected,
        g
    );
put(is_selected_map, aer.first, is_selected);
return aer;
}
```

When having added a new (abstract) edge to the graph, the edge descrip-

tor is used to set the my_custom_edge and the selectedness in the graph its my_custom_edge and is_selected map .

Here is the demo:

Algorithm 306 Demo of ‘add_custom_and_selectable_vertex’

```
#include <cassert>
#include "add_custom_and_selectable_edge.h"
#include "
    create_empty_directed_custom_and_selectable_edges_and_vertices_graph
    .h"

#include "
    create_empty_undirected_custom_and_selectable_edges_and_vertices_graph
    .h"

void add_custom_and_selectable_edge_demo() noexcept
{
    auto g =
        create_empty_directed_custom_and_selectable_edges_and_vertices_graph
        ();
    assert(boost::num_vertices(g) == 0);
    assert(boost::num_edges(g) == 0);
    const auto vd_from = boost::add_vertex(g);
    const auto vd_to = boost::add_vertex(g);
    add_custom_and_selectable_edge(
        vd_from, vd_to,
        my_custom_edge("X"),
        true,
        g
    );
    assert(boost::num_edges(g) == 1);
}
```

18.5 Creating a Markov-chain with custom and selectable vertices

18.5.1 Graph

Figure 55 shows the graph that will be reproduced:

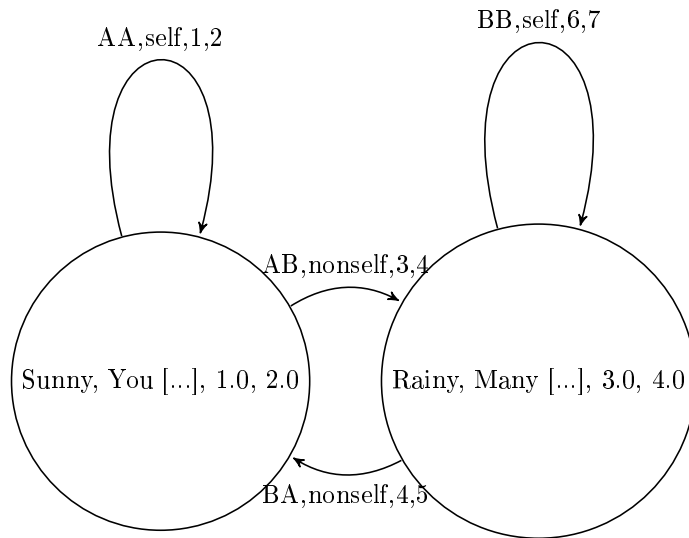


Figure 55: A two-state Markov chain where the edges and vertices have custom properties. The edges' and vertices' properties are nonsensical

18.5.2 Function to create such a graph

Here is the code creating a two-state Markov chain with custom edges and vertices:

18.5.3 Creating such a graph

Here is the demo:

Algorithm 308 Demo of the ‘create_custom_and_selectable_edges_and_vertices_markov_chain’ function (algorithm 307)

```
#include <cassert>
#include "
    create_custom_and_selectable_edges_and_vertices_markov_chain
    .h"
#include "get_vertex_selectednesses.h"

void
    create_custom_and_selectable_edges_and_vertices_markov_chain_demo
    () noexcept
{
    const auto g
        =
            create_custom_and_selectable_edges_and_vertices_markov_chain
            ();
    const std::vector<bool>
        expected_selectednesses{
            true, false
        };
    const std::vector<bool>
        vertex_selectednesses{
            get_vertex_selectednesses(g)
        };
    assert(expected_selectednesses
        == vertex_selectednesses
    );
}
```

18.5.4 The .dot file produced

Algorithm 309 .dot file created from the 'create_custom_and_selectable_vertices_markov_chain' function (algorithm 307), converted from graph to .dot file using algorithm 48

```
digraph G {
0[label="Sunny,Yellow$$$SPACE$$$thing,1,1", regular="1"];
1[label="Rainy,Grey$$$SPACE$$$things,3,3", regular="0"];
0->0 [label="A$$$SPACE$$$to$$$SPACE$$$A,,1,1", regular="1"];
0->1 [label="A$$$SPACE$$$to$$$SPACE$$$B,,1,1", regular="0"];
1->0 [label="B$$$SPACE$$$to$$$SPACE$$$A,,1,1", regular="0"];
1->1 [label="B$$$SPACE$$$to$$$SPACE$$$B,,1,1", regular="1"];
}
```

18.5.5 The .svg file produced

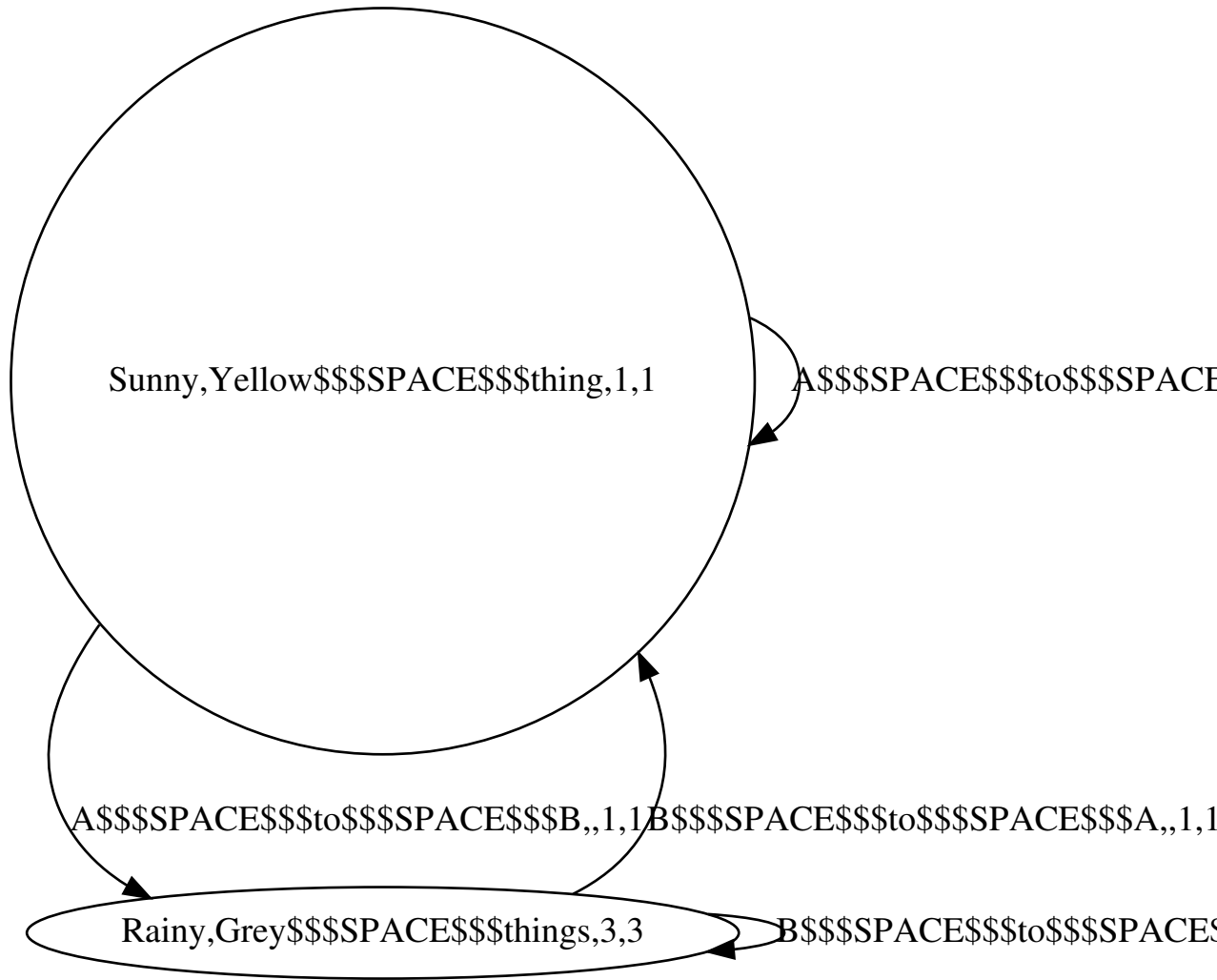


Figure 56: .svg file created from the ‘create_custom_and_selectable_vertices_markov_chain’ function (algorithm 220) its .dot file, converted from .dot file to .svg using algorithm 339

Note how the .svg changed its appearance due to the Graphviz ‘regular’ property (see chapter 25.2): the vertex labeled ‘Sunny’ is drawn according to the Graphviz ‘regular’ attribute, which makes it a circle. The other vertex, labeled ‘Rainy’ is not drawn as such and retained its ellipsoid appearance.

18.6 Creating K_2 with custom and selectable edges and vertices

18.6.1 Graph

We reproduce the K_2 with custom vertices of chapter 12.8 , but now are vertices can be selected as well:

[graph here]

18.6.2 Function to create such a graph

Algorithm 310 Creating K_3 as depicted in figure 30

```
#include "
    create_empty_undirected_custom_and_selectable_edges_and_vertices_graph
    .h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex,
        boost::property<
            boost::vertex_is_selected_t, bool
        >
    >,
    >,
    boost::property<
        boost::edge_custom_type_t, my_custom_edge,
        boost::property<
            boost::edge_is_selected_t, bool
        >
    >
>
create_custom_and_selectable_edges_and_vertices_k2_graph
() noexcept
{
    auto g
    =
        create_empty_undirected_custom_and_selectable_edges_and_vertices_graph
        ();
    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);
    const auto aer = boost::add_edge(vd_a, vd_b, g);
    assert(aer.second);

    auto my_custom_vertexes_map = get(
        boost::vertex_custom_type,
        g
    );
    put(my_custom_vertexes_map, vd_a,
        my_custom_vertex("A", "source", 0.0, 0.0)
    );
    put(my_custom_vertexes_map, vd_b,
        my_custom_vertex("B", "target", 3.14, 3.14)
    );

    auto is_selected_map = get344(
        boost::vertex_is_selected,
        g
    );
    put(is_selected_map, vd_a, true);
    put(is_selected_map, vd_b, false);
    return g;
}
```


Most of the code is a slight modification of algorithm 223. In the end, the associated `my_custom_vertex` and `is_selected` properties are obtained as `boost::property_maps` and set with the desired `my_custom_vertex` objects and `selectednesses`.

18.6.3 Creating such a graph

Here is the demo:

Algorithm 311 Demo of the ‘`create_custom_and_selectable_edges_and_vertices_k2_graph`’ function (algorithm 310)

```
#include <cassert>
#include "
    create_custom_and_selectable_edges_and_vertices_k2_graph
    .h"
#include "has_custom_vertex_with_my_vertex.h"

void
    create_custom_and_selectable_edges_and_vertices_k2_graph_demo
    () noexcept
{
    const auto g =
        create_custom_and_selectable_edges_and_vertices_k2_graph
        ();
    assert(boost::num_edges(g) == 1);
    assert(boost::num_vertices(g) == 2);
    assert(has_custom_vertex_with_my_custom_vertex(
        my_custom_vertex("A", "source", 0.0, 0.0), g)
    );
    assert(has_custom_vertex_with_my_custom_vertex(
        my_custom_vertex("B", "target", 3.14, 3.14), g)
    );
}
```

18.6.4 The .dot file produced

Algorithm 312 .dot file created from the ‘`create_custom_and_selectable_vertices_k2_graph`’ function (algorithm 310), converted from graph to .dot file using algorithm 48

```
graph G {
0[label="A,source,0,0", regular="1"];
1[label="B,target,3.14,3.14", regular="0"];
0--1 [label=",,1,1", regular="0"];
}
```

18.6.5 The .svg file produced

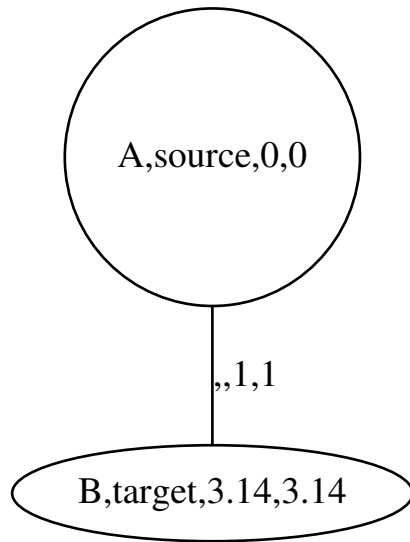


Figure 57: .svg file created from the ‘create_custom_and_selectable_vertices_k2_graph’ function (algorithm 220) its .dot file, converted from .dot file to .svg using algorithm 339

Note how the .svg changed its appearance due to the Graphviz ‘regular’ property (see chapter 25.2): the vertex labeled ‘A’ is drawn according to the Graphviz ‘regular’ attribute, which makes it a circle. The other vertex, labeled ‘B’ is not drawn as such and retained its ellipsoid appearance.

19 Working on graphs with custom and selectable edges and vertices

This chapter shows some basic operations to do on graphs with custom and selectable edges and vertices.

- Storing an directed/undirected graph with custom and selectable edges and vertices as a .dot file: chapter 19.1
- Loading a directed graph with custom and selectable edges and vertices from a .dot file: chapter 19.2
- Loading an undirected directed graph with custom and selectable edges and vertices from a .dot file: chapter 19.3

19.1 Storing a graph with custom and selectable edges and vertices as a .dot

If you used the ‘create_custom_and_selectable_edges_and_vertices_k2_graph’ function (algorithm 310) to produce a K_2 graph with edges and vertices associated with (1) my_custom_edge/my_custom_vertex objects, and (2) a boolean indicating its selectedness, you can store such graphs with algorithm 313:

Algorithm 313 Storing a graph with custom and selectable edges and vertices as a .dot file

```
#include <fstream>
#include <string>
#include <boost/graph/graphviz.hpp>
#include "install_edge_custom_type.h"
#include "install_edge_is_selected.h"
#include "install_vertex_custom_type.h"
#include "install_vertex_is_selected.h"
#include "make_custom_and_selectable_vertices_writer.h"
#include "my_custom_edge.h"
#include "my_custom_vertex.h"

template <typename graph>
void
    save_custom_and_selectable_edges_and_vertices_graph_to_dot
    (
        const graph& g,
        const std::string& filename
    )
{
    std::ofstream f(filename);
    boost::write_graphviz(f, g,
        make_custom_and_selectable_vertices_writer(
            get(boost::vertex_custom_type, g),
            get(boost::vertex_is_selected, g)
        ),
        make_custom_and_selectable_vertices_writer(
            get(boost::edge_custom_type, g),
            get(boost::edge_is_selected, g)
        )
    );
}
```

We re-use the writer.

Special about this, is that even for Graphviz-unfriendly input, it still works.

19.2 Loading a directed graph with custom and selectable edges and vertices from a .dot

When loading a graph from file, one needs to specify a type of graph. In this example, an directed graph with custom and selectable edges and vertices is loaded, as shown in algorithm 314:

Algorithm 314 Loading a directed graph with custom and selectable edges and vertices from a .dot file

```
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "
    create_empty_directed_custom_and_selectable_edges_and_vertices_graph
    .h"
#include "is_regular_file.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex,
        boost::property<
            boost::vertex_is_selected_t, bool
        >
    >,
    >,
    boost::property<
        boost::edge_custom_type_t, my_custom_edge,
        boost::property<
            boost::edge_is_selected_t, bool
        >
    >
>
load_directed_custom_and_selectable_edges_and_vertices_graph_from_dot
(
    const std::string& dot_filename
)
{
    assert(is_regular_file(dot_filename));
    std::ifstream f(dot_filename.c_str());
    auto g =
        create_empty_directed_custom_and_selectable_edges_and_vertices_graph
        ();
    boost::dynamic_properties dp(
        boost::ignore_other_properties
    );
    dp.property("label", get(boost::vertex_custom_type, g));
    ;
    dp.property("regular", get(boost::vertex_is_selected, g));
    );
    dp.property("label", get(boost::edge_custom_type, g));
    dp.property("regular", get(boost::edge_is_selected, g));
    ;
    boost::read_graphviz(f, g, dp);
    return g;
}
```

In this algorithm, first it is checked if the file to load exists. Then an empty directed graph is created. Then, a `boost::dynamic_properties` is created with its default constructor, after which

- The Graphviz attribute ‘node_id’ (see chapter 25.2 for most Graphviz attributes) is connected to a vertex its ‘my_custom_vertex’ property
- The Graphviz attribute ‘label’ is connected to a vertex its ‘my_custom_vertex’ property
- The Graphviz attribute ‘regular’ is connected to a vertex its ‘is_selected’ vertex property

Algorithm 315 shows how to use the ‘load_directed_custom_vertices_graph_from_dot’ function:

Algorithm 315 Demonstration of the ‘load_directed_custom_and_selectable_edges_and_vertices_graph_from_dot’ function

```
#include <cassert>
#include "
    create_custom_and_selectable_edges_and_vertices_markov_chain
    .h"
#include "is_regular_file.h"
#include "
    save_custom_and_selectable_edges_and_vertices_graph_to_dot
    .h"

void
load_directed_custom_and_selectable_edges_and_vertices_graph_from_dot_demo
() noexcept
{
    const auto g
    =
        create_custom_and_selectable_edges_and_vertices_markov_chain
        ();
    const std::string filename{
        "
            create_custom_and_selectable_edges_and_vertices_markov_chain
            .dot"
    };
    save_custom_and_selectable_edges_and_vertices_graph_to_dot
    (
        g,
        filename
    );
    assert(is_regular_file(filename));
}
```

This demonstration shows how the Markov chain is created using the ‘create_custom_vertices_markov_chain’ function (algorithm 220), saved and then checked to exist.

19.3 Loading an undirected graph with custom and selectable edges and vertices from a .dot

When loading a graph from file, one needs to specify a type of graph. In this example, an undirected graph with custom and selectable vertices is loaded, as shown in algorithm 316:

Algorithm 316 Loading an undirected graph with custom vertices from a .dot file

```
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "
    create_empty_undirected_custom_and_selectable_edges_and_vertices_graph
    .h"
#include "is_regular_file.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
        boost::vertex_custom_type_t, my_custom_vertex,
        boost::property<
            boost::vertex_is_selected_t, bool
        >
    >,
    >,
    boost::property<
        boost::edge_custom_type_t, my_custom_edge,
        boost::property<
            boost::edge_is_selected_t, bool
        >
    >
>
load_undirected_custom_and_selectable_edges_and_vertices_graph_from_dot
(
    const std::string& dot_filename
)
{
    assert(is_regular_file(dot_filename));
    std::ifstream f(dot_filename.c_str());
    auto g =
        create_empty_undirected_custom_and_selectable_edges_and_vertices_graph
        ();
    boost::dynamic_properties dp(
        boost::ignore_other_properties
    );
    dp.property("label", get(boost::vertex_custom_type, g));
    ;
    dp.property("regular", get(boost::vertex_is_selected, g));
    );
    dp.property("label", get(boost::edge_custom_type, g));
    dp.property("regular", get(boost::edge_is_selected, g));
    ;
    boost::read_graphviz(f, g, dp);
    return g;
}
```

The only difference with loading a directed graph, is that the initial empty graph is undirected instead. Chapter 19.2 describes the rationale of this function.

Algorithm 317 shows how to use the ‘load_undirected_custom_vertices_graph_from_dot’ function:

Algorithm 317 Demonstration of the ‘load_undirected_custom_and_selectable_edges_and_vertices_graph_from_dot’ function

```
#include <cassert>
#include "
    create_custom_and_selectable_edges_and_vertices_k2_graph
    .h"
#include "is_regular_file.h"
#include "
    save_custom_and_selectable_edges_and_vertices_graph_to_dot
    .h"

void
load_undirected_custom_and_selectable_edges_and_vertices_graph_from_dot_demo
() noexcept
{
    const auto g
    =
        create_custom_and_selectable_edges_and_vertices_k2_graph
        ();
    const std::string filename{
        "
            create_custom_and_selectable_edges_and_vertices_k2_graph
            .dot"
    };
    save_custom_and_selectable_edges_and_vertices_graph_to_dot
    (
        g,
        filename
    );
    assert(is_regular_file(filename));
}
```

This demonstration shows how K_2 with custom vertices is created using the ‘create_custom_vertices_k2_graph’ function (algorithm 223), saved and then checked to exist.

20 Building graphs with a graph name

Up until now, the graphs created have had no properties themselves. Sure, the edges and vertices have had properties, but the graph itself has had none. Until now.

In this chapter, graphs will be created with a graph name of type `std::string`

- An empty directed graph with a graph name: see chapter
- An empty undirected graph with a graph name: see chapter
- A two-state Markov chain with a graph name: see chapter
- K_3 with a graph name: see chapter

In the process, some basic (sometimes bordering trivial) functions are shown:

- Getting a graph its name: see chapter
- Setting a graph its name: see chapter

20.1 Create an empty directed graph with a graph name property

Algorithm 318 shows the function to create an empty directed graph with a graph name.

Algorithm 318 Creating an empty directed graph with a graph name

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

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    boost::no_property,
    boost::no_property,
    boost::property<
        boost::graph_name_t, std::string
    >
>
>
create_empty_directed_graph_with_graph_name() noexcept
{
    return {};
}
```

This `boost::adjacency_list` is of the following type:

- the first ‘boost::vecS’: select (that is what the ‘S’ means) that out edges are stored in a std::vector. This is the default way.
- the second ‘boost::vecS’: select that the graph vertices are stored in a std::vector. This is the default way.
- ‘boost::directedS’: select that the graph is directed. This is the default selectedness
- the first ‘boost::no_property’: the vertices have no properties. This is the default (non-)property
- the second ‘boost::no_property’: the vertices have no properties. This is the default (non-)property
- ‘boost::property<boost::graph_name_t, std::string>’: the graph itself has a single property: its boost::graph_name has type std::string

Algorithm 319 demonstrates the ‘create_empty_directed_graph_with_graph_name’ function.

Algorithm 319 Demonstration of ‘create_empty_directed_graph_with_graph_name’

```
#include <cassert>
#include "create_empty_directed_graph_with_graph_name.h"

void create_empty_directed_graph_with_graph_name_demo()
    noexcept
{
    auto g
        = create_empty_directed_graph_with_graph_name();
    assert(boost::num_edges(g) == 0);
    assert(boost::num_vertices(g) == 0);
}
```

20.2 Create an empty undirected graph with a graph name property

Algorithm 320 shows the function to create an empty undirected graph with a graph name.

Algorithm 320 Creating an empty undirected graph with a graph name

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

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::no_property,
    boost::no_property,
    boost::property<
        boost::graph_name_t, std::string
    >
>
>
create_empty_undirected_graph_with_graph_name() noexcept
{
    return {};
}
```

This code is very similar to the code described in chapter 318, except that the directedness (the third template argument) is undirected (due to the `boost::undirectedS`).

Algorithm 321 demonstrates the ‘`create_empty_undirected_graph_with_graph_name`’ function.

Algorithm 321 Demonstration of ‘`create_empty_undirected_graph_with_graph_name`’

```
#include <cassert>

#include "create_empty_undirected_graph_with_graph_name.h"

void create_empty_undirected_graph_with_graph_name_demo()
    noexcept
{
    auto g = create_empty_undirected_graph_with_graph_name
        ();
    assert(boost::num_edges(g) == 0);
    assert(boost::num_vertices(g) == 0);
}
```

20.3 Get a graph its name property

Algorithm 322 Get a graph its name

```
#include <string>
#include <boost/graph/properties.hpp>

template <typename graph>
std::string get_graph_name(
    const graph& g
) noexcept
{
    return get_property(
        g, boost::graph_name
    );
}
```

Algorithm 323 demonstrates the ‘get_graph_name’ function.

Algorithm 323 Demonstration of ‘get_graph_name’

```
#include <cassert>

#include "create_empty_directed_graph_with_graph_name.h"
#include "get_graph_name.h"
#include "set_graph_name.h"

void get_graph_name_demo() noexcept
{
    auto g = create_empty_directed_graph_with_graph_name();
    const std::string name{"Dex"};
    set_graph_name(name, g);
    assert(get_graph_name(g) == name);
}
```

20.4 Set a graph its name property

Algorithm 324 Set a graph its name

```
#include <cassert>
#include <string>
#include <boost/graph/properties.hpp>

template <typename graph>
void set_graph_name(
    const std::string& name,
    graph& g
) noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const");
    get_property(g, boost::graph_name) = name;
}
```

Algorithm 325 demonstrates the ‘set_graph_name’ function.

Algorithm 325 Demonstration of ‘set_graph_name’

```
#include <cassert>

#include "create_empty_directed_graph_with_graph_name.h"
#include "get_graph_name.h"
#include "set_graph_name.h"

void set_graph_name_demo() noexcept
{
    auto g = create_empty_directed_graph_with_graph_name();
    const std::string name{"Dex"};
    set_graph_name(name, g);
    assert(get_graph_name(g) == name);
}
```

20.5 Create a directed graph with a graph name property

20.5.1 Graph

See figure 6.

20.5.2 Function to create such a graph

Algorithm 326 shows the function to create an empty directed graph with a graph name.

Algorithm 326 Creating a two-state Markov chain with a graph name

```
#include <cassert>
#include "create_empty_directed_graph_with_graph_name.h"
#include "set_graph_name.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    boost::no_property,
    boost::no_property,
    boost::property<boost::graph_name_t, std::string>
>
create_markov_chain_with_graph_name() noexcept
{
    auto g = create_empty_directed_graph_with_graph_name();
    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);
    const auto aer_aa = boost::add_edge(vd_a, vd_a, g);
    assert(aer_aa.second);
    const auto aer_ab = boost::add_edge(vd_a, vd_b, g);
    assert(aer_ab.second);
    const auto aer_ba = boost::add_edge(vd_b, vd_a, g);
    assert(aer_ba.second);
    const auto aer_bb = boost::add_edge(vd_b, vd_b, g);
    assert(aer_bb.second);

    set_graph_name("Two-state_Markov_chain", g);
    return g;
}
```

20.5.3 Creating such a graph

Algorithm 327 demonstrates the ‘create_markov_chain_with_graph_name’ function.

Algorithm 327 Demonstration of ‘create_markov_chain_with_graph_name’

```
#include <cassert>
#include "create_markov_chain_with_graph_name.h"
#include "get_graph_name.h"

void create_markov_chain_with_graph_name_demo() noexcept
{
    const auto g = create_markov_chain_with_graph_name();
    assert(boost::num_vertices(g) == 2);
    assert(boost::num_edges(g) == 4);
    assert(get_graph_name(g) == "Two-state Markov chain");
}
```

20.5.4 The .dot file produced

Algorithm 328 .dot file created from the ‘create_markov_chain_with_graph_name’ function (algorithm 326), converted from graph to .dot file using algorithm 48

```
digraph G {
name="Two-state Markov chain";
0;
1;
0->0 ;
0->1 ;
1->0 ;
1->1 ;
}
```

20.5.5 The .svg file produced

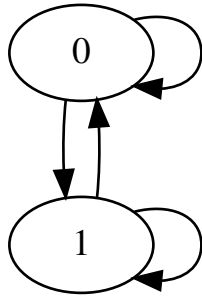


Figure 58: .svg file created from the ‘create_markov_chain_with_graph_name’ function (algorithm 326) its .dot file, converted from .dot file to .svg using algorithm 339

20.6 Create an undirected graph with a graph name property

20.6.1 Graph

See figure 8.

20.6.2 Function to create such a graph

Algorithm 329 shows the function to create K2 graph with a graph name.

Algorithm 329 Creating a K2 graph with a graph name

```
#include "create_empty_undirected_graph_with_graph_name.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::no_property,
    boost::no_property,
    boost::property<boost::graph_name_t, std::string>
>
create_k2_graph_with_graph_name() noexcept
{
    auto g = create_empty_undirected_graph_with_graph_name
        ();
    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);
    const auto aer = boost::add_edge(vd_a, vd_b, g);
    assert(aer.second);

    get_property(g, boost::graph_name) = "K2";

    return g;
}
```

20.6.3 Creating such a graph

Algorithm 330 demonstrates the ‘create_k2_graph_with_graph_name’ function.

Algorithm 330 Demonstration of ‘create_k2_graph_with_graph_name’

```
#include <cassert>

#include "create_k2_graph_with_graph_name.h"
#include "get_graph_name.h"

void create_k2_graph_with_graph_name_demo() noexcept
{
    const auto g = create_k2_graph_with_graph_name();
    assert(boost::num_vertices(g) == 2);
    assert(boost::num_edges(g) == 1);
    assert(get_graph_name(g) == "K2");
}
```

20.6.4 The .dot file produced

Algorithm 331 .dot file created from the ‘create_k2_graph_with_graph_name’ function (algorithm 329), converted from graph to .dot file using algorithm 48

```
graph G {
name="K2";
0;
1;
0--1 ;
}
```

20.6.5 The .svg file produced

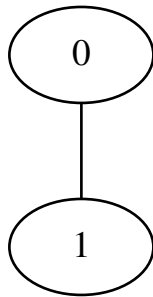


Figure 59: .svg file created from the ‘create_k2_graph_with_graph_name’ function (algorithm 329) its .dot file, converted from .dot file to .svg using algorithm 339

21 Working on graphs with a graph name

21.1 Storing a graph with a graph name property as a .dot file

This works:

Algorithm 332 Storing a graph with a graph name as a .dot file

```
#include <string>
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include <boost/graph/properties.hpp>

#include "get_graph_name.h"

template <typename graph>
void save_graph_with_graph_name_to_dot(
    const graph& g,
    const std::string& filename
)
{
    std::ofstream f(filename);
    boost::write_graphviz(
        f,
        g,
        boost::default_writer(),
        boost::default_writer(),
        //Unsure if this results in a graph
        //that can be loaded correctly
        //from a .dot file
        [g](std::ostream& os) {
            os << "name=\""
                << get_graph_name(g)
                << "\";\n";
        }
    );
}
```

21.2 Loading a directed graph with a graph name property from a .dot file

This will result in a directed graph with a name:

Algorithm 333 Loading a directed graph with a graph name from a .dot file

```
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "is_regular_file.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    boost::no_property,
    boost::no_property,
    boost::property<
        boost::graph_name_t, std::string
    >
>
>
load_directed_graph_with_graph_name_from_dot(
    const std::string& dot_filename
)
{
    using graph = boost::adjacency_list<
        boost::vecS,
        boost::vecS,
        boost::directedS,
        boost::no_property,
        boost::no_property,
        boost::property<
            boost::graph_name_t, std::string
        >
    >
>;

    assert(is_regular_file(dot_filename));

    graph g;

    boost::ref_property_map<graph*, std::string>
    graph_name{
        get_property(g, boost::graph_name)
    };
    boost::dynamic_properties dp{
        boost::ignore_other_properties
    };
    dp.property("name", graph_name);

    std::ifstream f(dot_filename.c_str());
    boost::read_graphviz(f, g, dp);
    return g;
}
```

21.3 Loading an undirected graph with a graph name property from a .dot file

This will result in an undirected graph with a name:

Algorithm 334 Loading an undirected graph with a graph name from a .dot file

```
#include <fstream>
#include <string>
#include <boost/graph/graphviz.hpp>
#include "create_empty_undirected_graph_with_graph_name.h"
"
#include "is_regular_file.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::no_property,
    boost::no_property,
    boost::property<
        boost::graph_name_t, std::string
    >
>
>
load_undirected_graph_with_graph_name_from_dot(
    const std::string& dot_filename
)
{
    using graph = boost::adjacency_list<
        boost::vecS,
        boost::vecS,
        boost::undirectedS,
        boost::no_property,
        boost::no_property,
        boost::property<
            boost::graph_name_t, std::string
        >
    >
    >;

    assert(is_regular_file(dot_filename));
    graph g;

    boost::ref_property_map<graph*, std::string>
    graph_name{
        get_property(g, boost::graph_name)
    };
    boost::dynamic_properties dp{
        boost::ignore_other_properties
    };
    dp.property("name", graph_name);

    std::ifstream f(dot_filename.c_str());
    boost::read_graphviz(f, g, dp);
    return g;
}
```

22 Other graph functions

Some functions that did not fit in.

22.1 Encode a `std::string` to a Graphviz-friendly format

You may want to use a label with spaces, comma's and/or quotes. Saving and loading these, will result in problem. This function replaces these special characters by a rare combination of ordinary characters.

Algorithm 335 Encode a `std::string` to a Graphviz-friendly format

```
#include <boost/algorithm/string/replace.hpp>

std::string graphviz_encode(std::string s) noexcept
{
    boost::algorithm::replace_all(s, ",", "$$$COMMA$$$");
    boost::algorithm::replace_all(s, "_", "$$$SPACE$$$");
    boost::algorithm::replace_all(s, "\"", "$$$QUOTE$$$");
    return s;
}
```

22.2 Decode a `std::string` from a Graphviz-friendly format

This function undoes the ‘`graphviz_encode`’ function (algorithm 335) and thus converts a Graphviz-friendly `std::string` to the original human-friendly `std::string`.

Algorithm 336 Decode a `std::string` from a Graphviz-friendly format to a human-friendly format

```
#include <boost/algorithm/string/replace.hpp>

std::string graphviz_decode(std::string s) noexcept
{
    boost::algorithm::replace_all(s, "$$$COMMA$$$", ",");
    boost::algorithm::replace_all(s, "$$$SPACE$$$", "_");
    boost::algorithm::replace_all(s, "$$$QUOTE$$$", "\"");
    return s;
}
```

22.3 Check if a `std::string` is Graphviz-friendly

There are pieces where I check if a `std::string` is Graphviz-friendly. This is done only where it matters. If it is tested not to matter, ‘`is_graphviz_friendly`’ is absent.

Algorithm 337 Check if a `std::string` is Graphviz-friendly

```
#include "graphviz_encode.h"

bool is_graphviz_friendly(const std::string& s) noexcept
{
    return graphviz_encode(s) == s;
}
```

23 Misc functions

These are some function I needed for creating this tutorial. Although they are not important for working with graphs, I used these heavily. These functions may be compiler-dependent, platform-dependent and/or there may be superior alternatives. I just add them for completeness.

23.1 Getting a data type as a `std::string`

This function will only work under GCC. I found this code at: <http://stackoverflow.com/questions/1055452/c-get-name-of-type-in-template>. Thanks to 'm-dudley' (Stack Overflow userpage at <http://stackoverflow.com/users/111327/m-dudley>).

Algorithm 338 Getting a data type its name as a `std::string`

```
#include <cstdlib>
#include <string>
#include <typeinfo>
#include <cxxabi.h>

template<typename T>
std::string get_type_name() noexcept
{
    std::string tname = typeid(T).name();
    int status = -1;
    char * const demangled_name{
        abi::__cxa_demangle(
            tname.c_str(), NULL, NULL, &status
        )
    };
    if(status == 0) {
        tname = demangled_name;
        std::free(demangled_name);
    }
    return tname;
}
```

23.2 Convert a .dot to .svg

All illustrations in this tutorial are created by converting .dot to a .svg (‘Scalable Vector Graphic’) file. This function assumes the program ‘dot’ is installed, which is part of Graphviz.

Algorithm 339 Convert a .dot file to a .svg

```
#include <cassert>
#include <string>
#include <sstream>
#include "has_dot.h"
#include "is_regular_file.h"
#include "is_valid_dot_file.h"

void convert_dot_to_svg(
    const std::string& dot_filename,
    const std::string& svg_filename
)
{
    assert(has_dot());
    assert(is_valid_dot_file(dot_filename));
    std::stringstream cmd;
    cmd << "dot-Tsvg_" << dot_filename << "_o_" <<
        svg_filename;
    std::system(cmd.str().c_str());
    assert(is_regular_file(svg_filename));
}
```

‘convert_dot_to_svg’ makes a system call to the program ‘dot’ to convert the .dot file to an .svg file.

23.3 Check if a file exists

Not the most smart way perhaps, but it does only use the STL.

Algorithm 340 Check if a file exists

```
#include <fstream>

bool is_regular_file(const std::string& filename)
    noexcept
{
    std::fstream f;
    f.open(filename.c_str(), std::ios::in);
    return f.is_open();
}
```

24 Errors

Some common errors.

24.1 Formed reference to void

This compile-time error occurs when you create a graph without a certain property, then subsequently reading that property, as in algorithm 341:

Algorithm 341 Creating the error ‘formed reference to void’

```
#include "create_k2_graph.h"
#include "get_vertex_names.h"

void formed_reference_to_void() noexcept
{
    get_vertex_names(create_k2_graph());
}
```

In algorithm 341 a graph is created with vertices of no properties. Then the names of these vertices, which do not exist, are tried to be read. If you want to read the names of the vertices, supply a graph that has this property.

24.2 No matching function for call to ‘clear_out_edges’

This compile-time error occurs when you want to clear the outward edges from a vertex in an undirected graph.

Algorithm 342 Creating the error ‘no matching function for call to clear_out_edges’

```
#include "create_k2_graph.h"

void no_matching_function_for_call_to_clear_out_edges()
    noexcept
{
    auto g = create_k2_graph();
    const auto vd = *vertices(g).first;
    boost::clear_in_edges(vd, g);
}
```

In algorithm 342 an undirected graph is created, a vertex descriptor is obtained, then its out edges are tried to be cleared. Either use a directed graph (which has out edges), or use the ‘boost::clear_vertex’ function instead.

24.3 No matching function for call to ‘clear_in_edges’

See chapter 24.2.

24.4 Undefined reference to boost::detail::graph::read_graphviz_new

You will have to link against the Boost.Graph and Boost.Regex libraries. In Qt Creator, this is achieved by adding these lines to your Qt Creator project file:

```
LIBS += -lboost_graph -lboost_regex
```

24.5 Property not found: node_id

When loading a graph from file (as in chapter 3.9) you will be using `boost::read_graphviz`. `boost::read_graphviz` needs a third argument, of type `boost::dynamic_properties`. When a graph does not have properties, do not use a default constructed version, but initialize with ‘`boost::ignore_other_properties`’ as a constructor argument instead. Algorithm 343 shows how to trigger this run-time error.

Algorithm 343 Creating the error ‘Property not found: node_id’

```
#include <cassert>
#include <fstream>
#include "is_regular_file.h"
#include "create_empty_undirected_graph.h"
#include "create_k2_graph.h"
#include "save_graph_to_dot.h"

void property_not_found_node_id() noexcept
{
    const std::string dot_filename{"
        property_not_found_node_id.dot"};
    // Create a file
    {
        const auto g = create_k2_graph();
        save_graph_to_dot(g, dot_filename);
        assert(is_regular_file(dot_filename));
    }

    // Try to read that file
    std::ifstream f(dot_filename.c_str());
    auto g = create_empty_undirected_graph();

    // Line below should have been
    // boost::dynamic_properties dp(boost::
    ignore_other_properties);
    boost::dynamic_properties dp; // Error

    try {
        boost::read_graphviz(f, g, dp);
    }
    catch (std::exception&) {
        return; // Should get here
    }
    assert(!"Should_not_get_here");
}
```

24.6 Stream zeroes

When loading a graph from a .dot file, in operator>>, I encountered reading zeroes, where I expected an XML formatted string:

```
std::istream& ribi::cmap::operator>>(std::istream& is, my_class& any_class) noexcept
{
    std::string s;
```

```

    is >> s; //s has an XML format
    assert(s != "0");
    any_class = my_class(s);
    return is;
}

```

This was because I misconfigured the reader. I did (heavily simplified code):

```

graph load_from_dot(const std::string& dot_filename)
{
    std::ifstream f(dot_filename.c_str());
    graph g;
    boost::dynamic_properties dp;
    dp.property("node_id", get(boost::vertex_custom_type, g));
    dp.property("label", get(boost::vertex_custom_type, g));
    boost::read_graphviz(f,g,dp);
    return g;
}

```

Where it should have been:

```

graph load_from_dot(const std::string& dot_filename)
{
    std::ifstream f(dot_filename.c_str());
    graph g;
    boost::dynamic_properties dp(boost::ignore_other_properties);
    dp.property("label", get(boost::vertex_custom_type, g));
    boost::read_graphviz(f,g,dp);
    return g;
}

```

The explanation is that by setting the boost::dynamic_property ‘node_id’ to ‘boost::vertex_custom_type’, operator>> will receive the node indices.

An alternative, but less clean solution, is to let operator>> ignore the node indices:

```

std::istream& ribi::cmap::operator>>(std::istream& is, my_class& any_class) noexcept
{
    std::string s;
    is >> s; //s has an XML format
    if (!is_xml(s)) { //Ignore node index
        any_class_class = my_class();
    }
    else {
        any_class_class = my_class(s);
    }
    return is;
}

```

25 Appendix

25.1 List of all edge, graph and vertex properties

The following list is obtained from the file ‘boost/graph/properties.hpp’.

| Edge | Graph | Vertex |
|------------------------|---------------|-----------------------|
| edge_all | graph_all | vertex_all |
| edge_bundle | graph_bundle | vertex_bundle |
| edge_capacity | graph_name | vertex_centrality |
| edge_centrality | graph_visitor | vertex_color |
| edge_color | | vertex_current_degree |
| edge_discover_time | | vertex_degree |
| edge_finished | | vertex_discover_time |
| edge_flow | | vertex_distance |
| edge_global | | vertex_distance2 |
| edge_index | | vertex_finish_time |
| edge_local | | vertex_global |
| edge_local_index | | vertex_in_degree |
| edge_name | | vertex_index |
| edge_owner | | vertex_index1 |
| edge_residual_capacity | | vertex_index2 |
| edge_reverse | | vertex_local |
| edge_underlying | | vertex_local_index |
| edge_update | | vertex_lowpoint |
| edge_weight | | vertex_name |
| edge_weight2 | | vertex_out_degree |
| | | vertex_owner |
| | | vertex_potential |
| | | vertex_predecessor |
| | | vertex_priority |
| | | vertex_rank |
| | | vertex_root |
| | | vertex_underlying |
| | | vertex_update |

25.2 Graphviz attributes

List created from www.graphviz.org/content/attrs, where only the attributes that are supported by all formats are listed:

| Edge | Graph | Vertex |
|----------------|---------------|---------------|
| arrowhead | _background | color |
| arrowsize | bgcolor | colorscheme |
| arrowtail | center | comment |
| color | charset | distortion |
| colorscheme | color | fillcolor |
| comment | colorscheme | fixedsize |
| decorate | comment | fontcolor |
| dir | concentrate | fontname |
| fillcolor | fillcolor | fontsize |
| fontcolor | fontcolor | gradientangle |
| fontname | fontname | height |
| fontsize | fontpath | image |
| gradientangle | fontsize | imagescale |
| headclip | forcelabels | label |
| headlabel | gradientangle | labelloc |
| headport | imagepath | layer |
| label | label | margin |
| labelangle | labeljust | nojustify |
| labeldistance | labelloc | orientation |
| labelfloat | landscape | penwidth |
| labelfontcolor | layerlistsep | peripheries |
| labelfontname | layers | pos |
| labelfontsize | layerselect | regular |
| layer | layersep | samplepoints |
| nojustify | layout | shape |
| penwidth | margin | shapefile |
| pos | nodesep | sides |
| style | nojustify | skew |
| tailclip | orientation | sortv |
| taillabel | outputorder | style |
| tailport | pack | width |
| weight | packmode | xlabel |
| xlabel | pad | z |
| | page | |
| | pagedir | |
| | penwidth | |
| | quantum | |
| | ratio | |
| | rotate | |
| | size | |
| | sortv | |
| | splines | |
| | style | |
| | viewport | |

References

- [1] Eckel Bruce. Thinking in c++, volume 1. 2002.
- [2] Marshall P Cline, Greg Lomow, and Mike Girou. *C++ FAQs*. Pearson Education, 1998.
- [3] Jarrod Hollingworth, Bob Swart, and Jamie Allsop. *C++ Builder 5 Developer's Guide with Cdrom*. Sams, 2000.
- [4] John Lakos. *Large-scale C++ software design*, volume 10. Addison-Wesley Reading, 1996.
- [5] Jesse Liberty. *Sams teach yourself C++ in 24 hours*. Sams Publishing, 2001.
- [6] Steve McConnell. *Code complete*. Pearson Education, 2004.
- [7] Scott Meyers. *Effective C++: 55 specific ways to improve your programs and designs*. Pearson Education, 2005.
- [8] Jeremy G Siek, Lie-Quan Lee, and Andrew Lumsdaine. *Boost Graph Library: User Guide and Reference Manual, The*. Pearson Education, 2001.
- [9] Bjarne Stroustrup. *The C++ Programming Language (3rd edition)*. 1997.
- [10] Bjarne Stroustrup. *The C++ Programming Language (4th edition)*. 2013.
- [11] Herb Sutter and Andrei Alexandrescu. *C++ coding standards: 101 rules, guidelines, and best practices*. Pearson Education, 2004.

Index

- #include, 18
- K_2 with named edges and vertices, create, 132
- K_2 with named vertices, create, 72
- K_2 , create, 35
- K_3 with named edges and vertices, create, 136
- K_3 , create, 38
- K_3 with named vertices, create, 75
- ‘demo’ function, 11
- ‘do’ function, 11
- [[:SPACE:]], 242, 243
- Add a vertex, 23
- Add an edge, 27
- Add bundled edge, 204
- Add bundled vertex, 171
- Add custom and selectable edge, 335
- Add custom and selectable vertex, 272
- Add custom edge, 304
- Add custom vertex, 237
- Add edge between named vertices, 100
- Add edge between two selected vertices, 286
- Add named edge, 123, 124
- Add named edge between vertices, 125, 126
- Add named vertex, 65, 66
- Add vertex, 23
- add_edge, 28
- aer_, 29
- All edge properties, 376
- All graph properties, 376
- All vertex properties, 376
- Alternative syntax for put, 67
- assert, 21, 28
- auto, 19
- boost::add_edge, 27, 28, 33, 36, 99, 124, 285
- boost::add_edge result, 29
- boost::add_vertex, 23, 33, 36
- boost::adjacency_list, 19, 64, 121, 122, 235
- boost::adjacency_matrix, 19
- boost::clear_in_edges, 97
- boost::clear_out_edges, 96
- boost::clear_vertex, 96
- boost::degree does not exist, 49
- boost::directedS, 20, 64, 121, 170, 235, 355
- boost::dynamic_properties, 60, 116, 164, 195, 227, 263, 292, 325, 350, 373
- boost::edge does not exist, 51
- boost::edge_bundled_type_t, 201
- boost::edge_custom_type, 304
- boost::edge_custom_type_t, 300
- boost::edge_name_t, 121, 122
- boost::edges does not exist, 29–31
- boost::get does not exist, 12, 66, 96
- boost::graph_name, 355
- boost::graph_name_t, 355
- boost::ignore_other_properties, 60, 373
- boost::in_degree does not exist, 49
- boost::isomorphism, 110, 259
- boost::make_label_writer, 113
- boost::no_property, 355
- boost::num_edges, 22
- boost::num_vertices, 21
- boost::out_degree does not exist, 49
- boost::property, 64, 121, 122, 201, 235, 300, 355
- boost::put does not exist, 66, 96
- boost::read_graphviz, 60, 116, 164, 195, 227, 263, 326, 373
- boost::remove_edge, 101, 155
- boost::remove_vertex, 98
- boost::undirectedS, 20, 65, 122, 171, 202, 236, 270, 302, 333, 356
- boost::vecS, 20, 63, 120, 122, 170, 235, 355
- boost::vertex_custom_type, 238
- boost::vertex_custom_type_t, 235
- boost::vertex_name, 66

| | |
|---|---|
| <code>boost::vertex_name_t</code> , 64, 121, 122 | Create an empty graph, 19 |
| <code>boost::vertices</code> does not exist, 25, 27, 31 | Create an empty graph with named edges and vertices, 122 |
| <code>boost::write_graphviz</code> , 58, 113 | Create an empty undirected graph with named vertices, 64 |
| <code>BOOST_INSTALL_PROPERTY</code> , 234, 267, 299, 330 | Create bundled edges and vertices K3 graph, 214 |
| <code>bundled_vertices_writer</code> , 192 | Create bundled edges and vertices Markov chain, 208 |
| Clear first vertex with name, 97 | Create bundled vertices K2 graph, 179 |
| <code>const</code> , 19 | Create bundled vertices Markov chain, 174 |
| <code>const-correctness</code> , 19 | Create custom and selectable edges and vertices K2 graph, 344 |
| Convert dot to svg, 371 | Create custom and selectable edges and vertices Markov chain, 338 |
| Count edges with selectedness, 322 | Create custom and selectable vertices K2 graph, 281 |
| Count vertices with selectedness, 284 | Create custom and selectable vertices Markov chain, 275 |
| Counting the number of edges, 22 | Create custom edges and vertices K3 graph, 312 |
| Counting the number of vertices, 21 | Create custom edges and vertices Markov chain, 308 |
| Create K_2 , 35 | Create custom vertices K2 graph, 244 |
| Create K_2 graph, 36 | Create custom vertices Markov chain, 241 |
| Create K_2 with named edges and vertices, 132 | Create custom vertices path graph, 247 |
| Create K_2 with named vertices, 72 | Create direct-neighbour named edges and vertices subgraph, 158 |
| Create K_3 , 38 | Create direct-neighbour named vertices subgraph, 106 |
| Create K_3 graph, 39 | Create direct-neighbour subgraph, 54 |
| Create K_3 with named edges and vertices, 136 | Create directed graph, 32 |
| Create K_3 with named vertices, 75 | Create directed graph from .dot, 59 |
| Create .dot from graph, 58 | Create directed graph with named edges and vertices from .dot, 163 |
| Create .dot from graph with bundled edges and vertices, 224 | Create directed graph with named vertices from .dot, 115 |
| Create .dot from graph with custom edges and vertices, 323 | Create empty directed bundled edges and vertices graph, 201 |
| Create .dot from graph with named edges and vertices, 161 | Create empty directed bundled vertices graph, 170 |
| Create .dot from graph with named vertices, 112 | Create empty directed custom and selectable edges and vertices graph, 331 |
| Create all direct-neighbour named edges and vertices subgraphs, 160 | |
| Create all direct-neighbour named vertices subgraphs, 108 | |
| Create all direct-neighbour subgraphs, 56 | |
| Create an empty directed graph, 18 | |
| Create an empty directed graph with named edges and vertices, 120 | |
| Create an empty directed graph with named vertices, 63 | |

| | |
|---|--|
| Create empty directed custom and selectable vertices graph, 268 | Create named edges and vertices Markov chain, 130 |
| Create empty directed custom edges and vertices graph, 300 | Create named edges and vertices path graph, 141 |
| Create empty directed custom vertices graph, 235 | Create named edges and vertices Petersen graph, 145 |
| Create empty directed graph, 18 | Create named vertices K2 graph, 73 |
| Create empty directed graph with graph name, 354 | Create named vertices Markov chain, 70 |
| Create empty directed named edges and vertices graph, 120 | Create named vertices path graph, 79 |
| Create empty directed named vertices graph, 63 | Create named vertices Petersen graph, 83 |
| Create empty undirected bundled edges and vertices graph, 202 | Create path graph, 40, 41 |
| Create empty undirected bundled vertices graph, 171 | Create path graph with custom vertices, 246 |
| Create empty undirected custom and selectable edges and vertices graph, 333 | Create path graph with named edges and vertices, 139 |
| Create empty undirected custom and selectable vertices graph, 270 | Create path graph with named vertices, 78 |
| Create empty undirected custom edges and vertices graph, 302 | Create Petersen graph, 43, 45 |
| Create empty undirected custom vertices graph, 236 | Create Petersen graph with named vertices, 81, 143 |
| Create empty undirected graph, 20 | Create undirected graph from .dot, 60 |
| Create empty undirected graph with graph name, 356 | Create undirected graph with bundled edges and vertices from .dot, 229 |
| Create empty undirected named edges and vertices graph, 122 | Create undirected graph with custom edges and vertices from .dot, 327 |
| Create empty undirected named vertices graph, 65 | Create undirected graph with named vertices from .dot, 117 |
| Create K2 graph with graph name, 362 | custom_and_selectable_vertices_writer, 290 |
| Create K3 vertices path graph, 76 | custom_vertex_invariant, 259 |
| Create Markov chain, 33 | Declaration, my_bundled_edge, 200 |
| Create Markov chain with graph name, 359 | Declaration, my_bundled_vertex, 169 |
| Create Markov chain with named edges and vertices, 129 | Declaration, my_custom_edge, 298 |
| Create Markov chain with named vertices, 69 | Declaration, my_custom_vertex, 233 |
| Create named edges and vertices K_2 graph, 134 | decltype(auto), 12 |
| Create named edges and vertices K3 graph, 137 | directed graph, 14 |
| | Directed graph, create, 32 |
| | ed_, 31 |
| | edge, 51 |
| | Edge descriptor, 30 |

| | |
|---|--|
| Edge descriptors, get, 31 | Get first vertex with name out degree, 91 |
| Edge iterator, 29 | Get graph name, 357 |
| Edge iterator pair, 29 | Get my_bundled_edge, 221 |
| Edge properties, 376 | Get my_custom_vertex, 254 |
| Edge, add, 27 | Get my_custom_vertexes, 239 |
| edge_is_selected, 330 | Get n edges, 22 |
| edge_is_selected_t, 330 | Get n vertices, 21 |
| edges, 29, 31 | Get type name, 370 |
| Edges, counting, 22 | Get vertex descriptors, 26 |
| eip_, 29 | Get vertex iterators, 25 |
| Empty directed graph with named edges and vertices, create, 120 | Get vertex name, 93 |
| Empty directed graph with named vertices, create, 63 | Get vertex names, 68 |
| Empty directed graph, create, 18 | Get vertex out degrees, 49 |
| Empty graph with named edges and vertices, create, 122 | Get vertices, 25 |
| Empty graph, create, 19 | get_edge_names, 128 |
| Empty undirected graph with named vertices, create, 64 | Graph properties, 376 |
| | Graphviz, 58 |
| | graphviz decode, 368 |
| | graphviz encode, 368 |
| Find first bundled edge with my_bundled_edge, 220 | Has bundled edge with my_bundled_edge, 218 |
| Find first bundled vertex with my_vertex, 186 | Has bundled vertex with my_vertex, 184 |
| Find first custom edge with my_custom_edge, 317 | Has custom edge with my_custom_edge, 315 |
| Find first custom vertex with my_vertex, 252 | Has custom vertex with my_vertex, 250 |
| Find first edge by name, 151 | Has edge between vertices, 51 |
| Find first vertex with name, 89, 104 | Has edge with name, 149 |
| Formed reference to void, 372 | Has vertex with name, 88 |
| | header file, 18 |
| get, 12, 66, 96, 238, 304 | idegree, 49 |
| Get bundled vertex my_bundled_vertex, 187 | in_degree, 49 |
| Get bundled vertex my_vertexes, 172 | Install edge custom type, 299 |
| Get custom edge my_custom_edge, 318 | Install edge_is_selected, 330 |
| Get custom vertex my_custom_vertex objects, 240 | Install vertex custom type, 234 |
| Get edge between vertices, 52 | Install vertex_is_selected, 267 |
| Get edge descriptors, 31 | Is isomorphic, 57, 111, 260 |
| Get edge iterators, 30 | Is regular file, 371 |
| Get edge my_bundled_edges, 206 | is_graphviz_friendly, 369 |
| Get edge my_custom_edges, 306 | link, 373 |
| Get edge name, 152 | Load directed bundled edges and vertices graph from dot, 226 |

| | |
|--|--|
| Load directed bundled vertices graph from dot, 194 | Markov chain with named edges and vertices, create, 129 |
| Load directed custom and selectable edges and vertices graph from dot, 349 | Markov chain with named vertices, create, 69 |
| Load directed custom edges and vertices graph from dot, 325, 365 | member, 169, 200, 233, 298 |
| Load directed custom vertices graph from dot, 263, 292 | my_bundled_edge, 200 |
| Load directed graph from .dot, 59 | my_bundled_edge declaration, 200 |
| Load directed graph from dot, 59 | my_bundled_edge.h, 200 |
| Load directed graph with named edges and vertices from .dot, 163 | my_bundled_vertex, 169, 170 |
| Load directed graph with named vertices from .dot, 115 | my_bundled_vertex.h, 169 |
| Load directed named edges and vertices graph from dot, 163 | my_custom_edge, 298 |
| Load directed named vertices graph from dot, 116 | my_custom_edge declaration, 298 |
| Load undirected bundled edges and vertices graph from dot, 230 | my_custom_edge.h, 298 |
| Load undirected bundled vertices graph from dot, 197 | my_custom_vertex, 233 |
| Load undirected custom edges and vertices graph from dot, 328, 367 | my_custom_vertex declaration, 233 |
| Load undirected custom vertices graph from dot, 265, 295, 352 | my_custom_vertex.h, 233 |
| Load undirected graph from .dot, 60 | my_edge, 201, 300 |
| Load undirected graph from _dot, 61 | my_vertex, 235 |
| Load undirected graph with bundled edges and vertices from .dot, 229 | my_vertex declaration, 169 |
| Load undirected graph with custom edges and vertices from .dot, 327 | Named edge, add, 123 |
| Load undirected graph with named vertices from .dot, 117 | Named edge, add between vertices, 125 |
| Load undirected named edges and vertices graph from dot, 166 | Named edges and vertices, create empty directed graph, 120 |
| Load undirected named vertices graph from dot, 118 | Named edges and vertices, create empty graph, 122 |
| m_, 169, 200, 233, 298 | Named vertex, add, 65 |
| macro, 234, 267, 299, 330 | Named vertices, create empty directed graph, 63 |
| make_bundled_vertices_writer, 191 | Named vertices, create empty undirected graph, 64 |
| make_custom_and_selectable_vertices_writer, 289 | named_vertex_invariant, 110 |
| | No matching function for call to clear_out_edges, 372 |
| | node_id, 373 |
| | noexcept, 18 |
| | noexcept specification, 18 |
| | Number of edges, get, 22 |
| | Number of vertices, get, 21 |
| | operator<, 258 |
| | out_degree, 49 |
| | Path graph with custom vertices, create, 246 |
| | Path graph with named edges and vertices, create, 139 |

| | |
|---|---|
| Path graph with named vertices, create, 78 | Set bundled edge my_bundled_edge, 223 |
| Path graph, create, 40 | Set bundled vertex my_bundled_vertexes, 190 |
| Petersen graph with named vertices, create, 81, 143 | Set edge name, 154 |
| Petersen graph, create, 43 | Set graph name, 358 |
| Property not found: node_id, 373, 374 | Set my_custom_edge, 320 |
| Property not found, 373 | Set my_custom_vertex, 255 |
| put, 66, 96 | Set my_custom_vertexes, 257 |
| put, alternative syntax, 67 | Set vertex my_vertex, 188 |
| | Set vertex name, 94 |
| read_graphviz_new, 373 | Set vertex names, 96 |
| read_graphviz_new, undefined reference, 373 | Set vertices names, 95 |
| Remove edge between vertices with names, 102 | static_assert, 23, 66 |
| Remove first edge with name, 156 | static_cast, 21 |
| Remove first vertex with name, 98 | std::copy, 27 |
| | std::count_if, 104, 284, 322 |
| | std::cout, 58 |
| | std::ifstream, 60 |
| S, 20, 355 | std::list, 19 |
| Save bundled edges and vertices graph to dot, 225 | std::ofstream, 58 |
| Save bundled vertices graph to dot, 191 | std::pair, 28 |
| Save custom and selectable edges and vertices graph to dot, 347 | std::vector, 19 |
| Save custom edges and vertices graph to dot, 324 | STL, 19 |
| Save custom vertices graph to dot, 262, 288 | Undefined reference to read_graphviz_new, 373 |
| Save graph as .dot, 58 | undirected graph, 14 |
| Save graph to dot, 58 | unsigned long, 21 |
| Save graph with bundled edges and vertices as .dot, 224 | vd, 28 |
| Save graph with custom edges and vertices as .dot, 323 | vd_, 24 |
| Save graph with graph name to dot, 364 | Vertex descriptor, 24, 27 |
| Save graph with name edges and vertices as .dot, 161 | Vertex descriptors, get, 26 |
| Save graph with named vertices as .dot, 112 | Vertex iterator, 25 |
| Save named edges and vertices graph to dot, 162 | Vertex iterator pair, 25 |
| Save named vertices graph to dot, 113 | Vertex iterators, get, 25 |
| Save named vertices graph to dot using lambda function, 114 | Vertex properties, 376 |
| | Vertex, add, 23 |
| | Vertex, add named, 65 |
| | vertex_custom_type, 232 |
| | vertex_is_selected, 267 |
| | vertex_is_selected_t, 267 |
| | vertices, 25, 27 |
| | Vertices, counting, 21 |
| | Vertices, set names, 95 |

vip_, 25