Graph and Path Searching Library

Andrew Grant (amg2215), Anton Igorevich (ain2108), Somya Vasudevan (sv2500)

Introduction

- Create a library that supports Adjacency List (DAG, DG, DT) and Matrix graph representations
- Make it super easy for users to use their own Vertex and Edge data structures
    - The library handles everything else under the hood
- Provide path finding algorithms that integrate with the library, but also work in other domains

Motivation

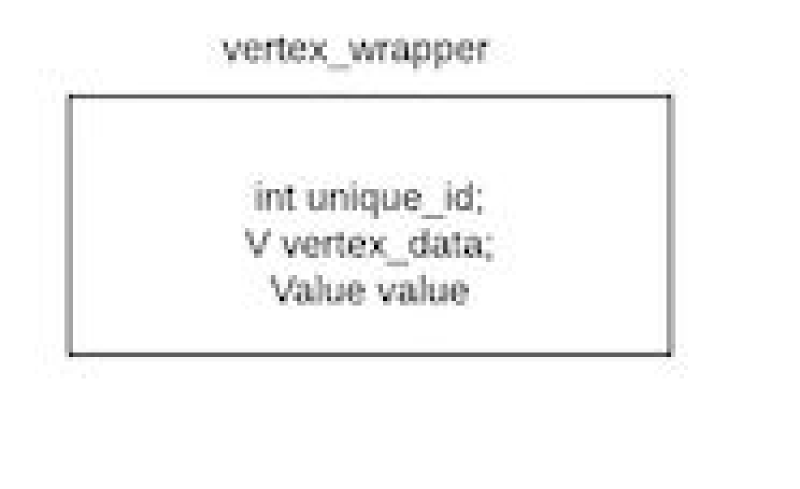
- Users want to create graphs and run algos on them with their own data types
- Idea: let’s make it really user for users to use their own data types
- Handle all the annoying details under the hood
- Create some very generic path algorithm code

Graph Library Implementation

- Choose between adjacency list and matrix
- Adjacency list: DAG, DG, DT
- Matrix: undirected
- All just structs

Graph Library Implementation contd.

- User provides vertex/edge data type
- Lib wraps vertex/edge in vertex_wrapper/edge_wrapper
- e.g.



vertex_wrapper

int unique_id;
V vertex_data;
Value value

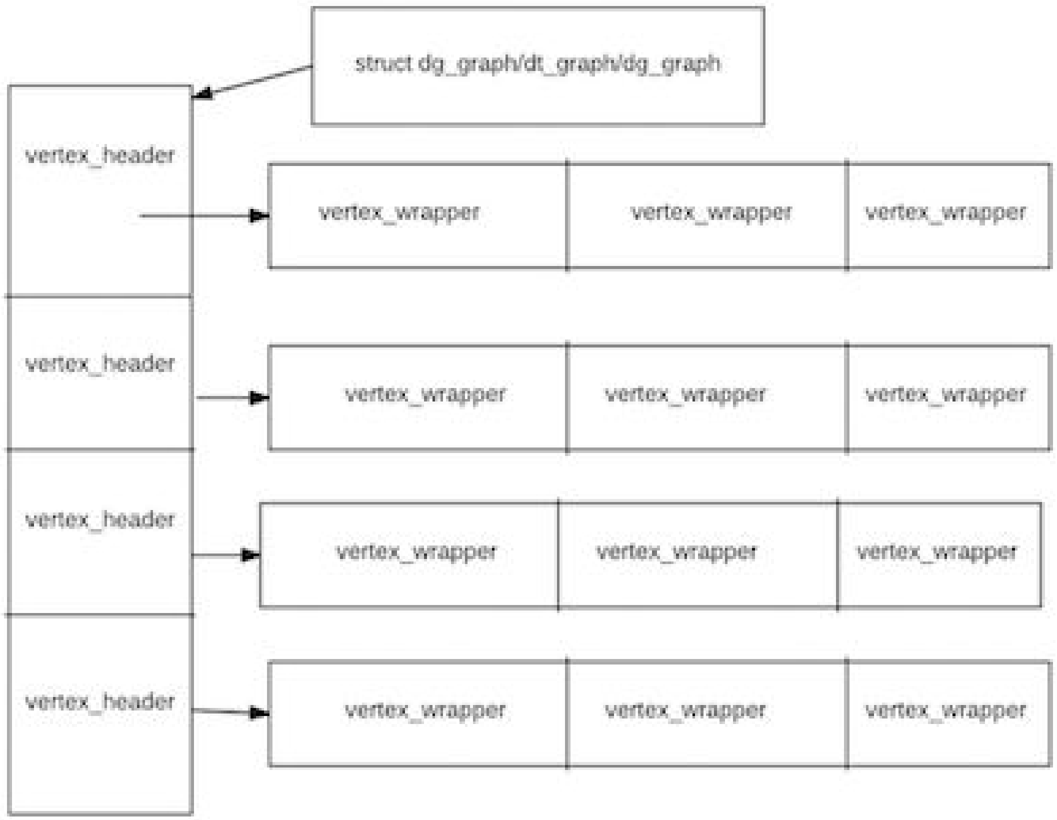
Graph Library Implementation contd. (interface)

- Interface is a bunch of functions
- Always at least provide underlying graph
- Other parameters are user defined vertex/edge
- Everything is a shared_ptr

Graph Library Implementation contd. (adjacency list)

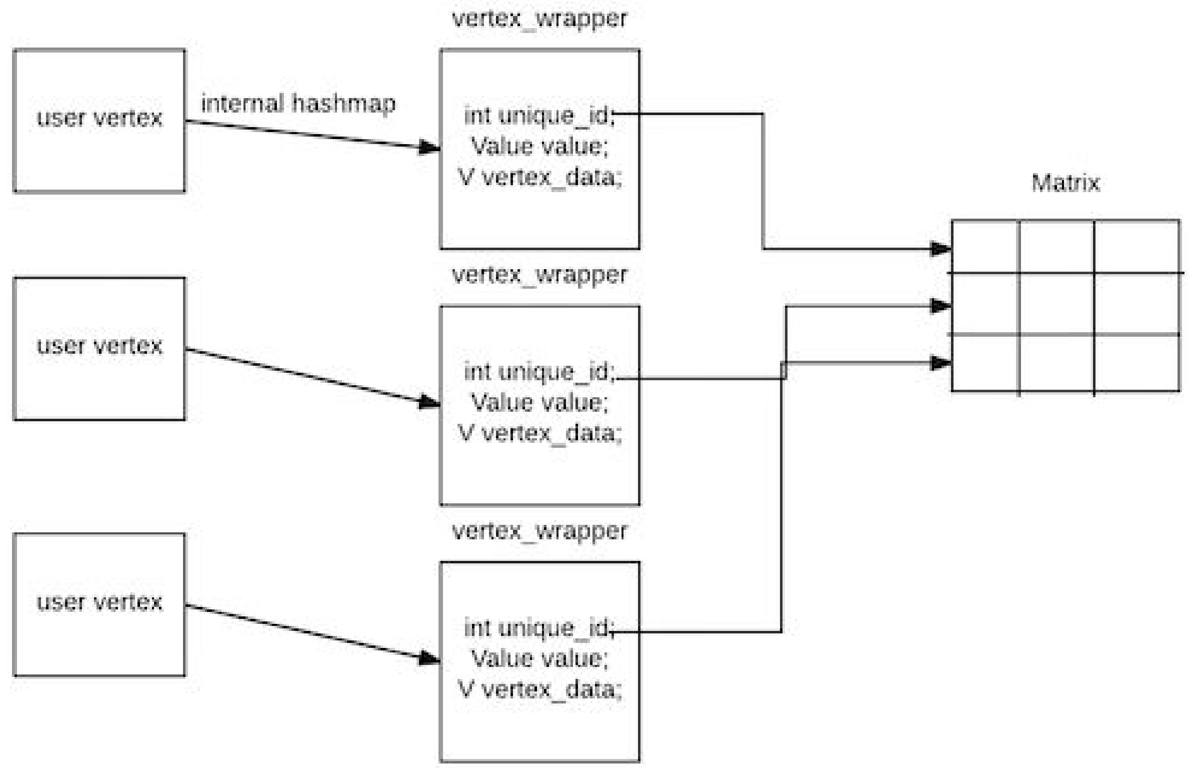
- Adjacency List
  - Another data type vertex_header
  - Vertex_header points to a vector of vertex_wrappers
  - Also points to a vector of edge_wrappers

Adjacency List Representation



Graph Library Implementation contd. (matrix)

- Use an internal id to use as index for each vertex_wrapper
- Maintain a hashmap from vertex to vertex_wrapper
    - Quickly get internal id for a vertex -> use as index intro matrix
- Use a vector of vector for underlying matrix


user vertex
internal hashmap
vertex_wrapper
int unique_id;
Value value;
V vertex_data;
Matrix
user vertex
vertex_wrapper
int unique_id;
Value value;
V vertex_data;
user vertex
vertex_wrapper
int unique_id;
Value value;
V vertex_data;

Path Algorithms

- Support BFS, DFS, UCS, A*, Dijkstra
- All run on the same code using templates
  - Inputs determine which algo is run
  - E.g. if the frontier set is a queue, we run bfs. If the frontier set is a stack, we run dfs

Path Algorithms Interface

- Very general interface
- Simply need to provide a start state, and a goal state
- We use concepts to make sure the user provides types that can work with the algorithms
    - The state must be hashable
    - The state must be comparable
    - The state must have expand() function that returns a vector of its children
    - The state must have a pointer to parent
    - The state must have a cost that is numeric

Path Algorithms (integration with Graph Library)

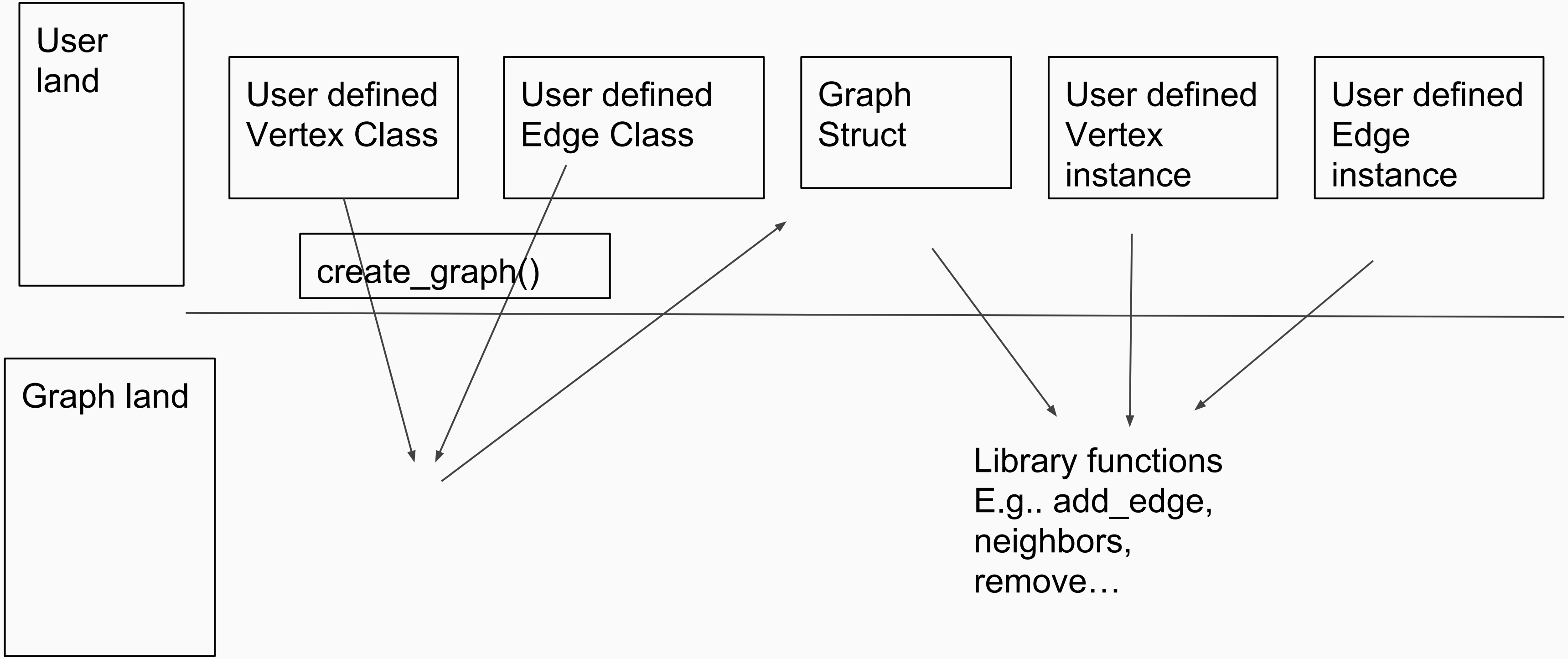
- The graph library doesn’t know anything about states
- But we still want the user to run the code we wrote for the path algos
- Create a wrapper under the hood

Using the Library

- The user simply must provide the library with his/her Vertex and Edges types
- Everything else is handled by the library
- Helper functions to create graphs

Using the Library contd.

- To create a graph
  - Select one of dt_graph, dag_graph, dg_graph, matrix_graph. Provide it with user defined vertex/edge data type
- API requires you send it graph pointer and user defined instances



User land

User defined Vertex Class

User defined Edge Class

Graph Struct

User defined Vertex instance

User defined Edge instance

create_graph()

Graph land

Library functions
E.g.. add_edge,
neighbors,
remove…

Concepts

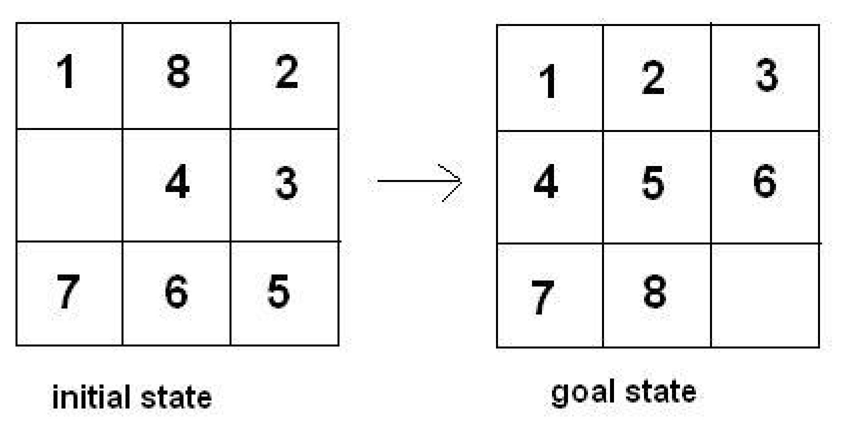
- We use concepts for two main reasons
- Reason 1: make debugging easier
  - Vertex/edge/state must adhere to certain properties
- Reason 2: function overloading
  - The same function name is used whether the user is working with a matrix or adjacency list
  - E.g. add_edge(g, e) can be used where g is a matrix or adjacency list

Memory Management

- Library works with smart pointers to avoid memory leaks
- Never uses new/delete
- Don’t allow user to point into underlying graph

Demo Time!

- 8 puzzle game
- (https://pravj.github.io/blog/development-story-of-puzzl/)



1	8	2
	4	3
7	6	5

initial state

→

1	2	3
4	5	6
7	8	

goal state