3_Exercices_fr

May 2, 2023

1 Exercice

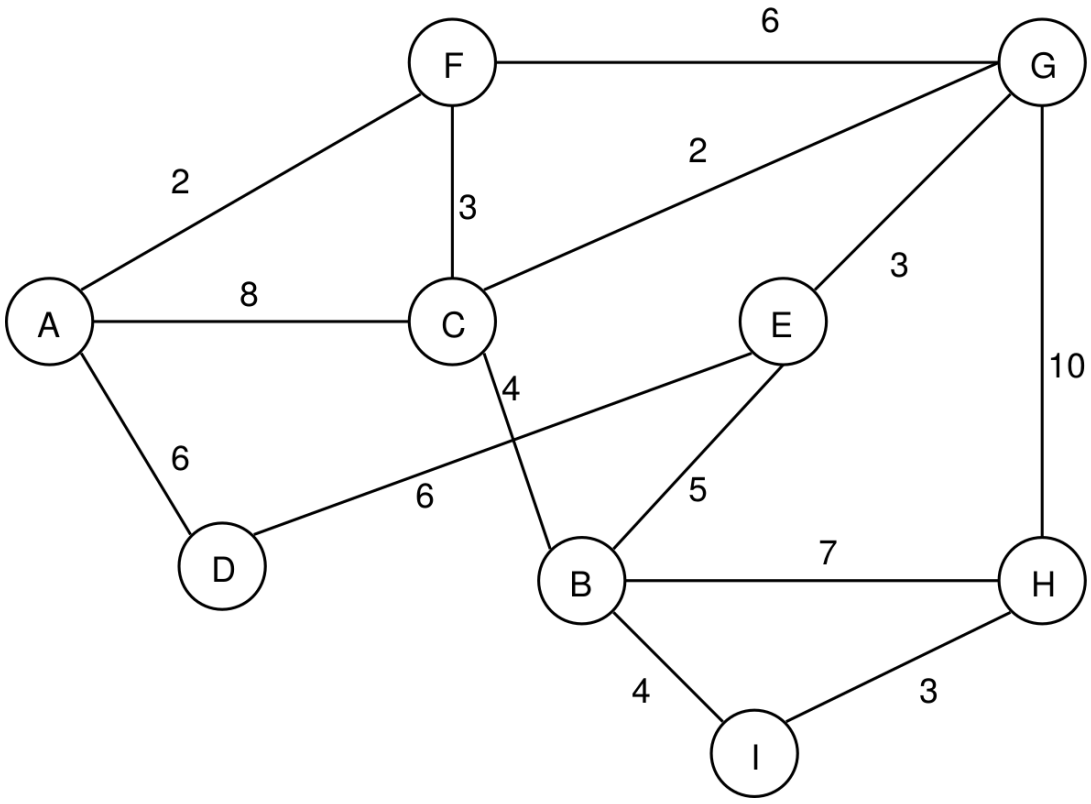
Construire un ensemble de structures pour représenter et manipuler des graphs (sommets et arrêtes).

- Ajouter la possibilité de naviguer le graphe de sommets en sommets en suivant les arrêtes.
- Ajouter la possibilité de mettre des données arbitraires sur les sommets et les arrêtes.
- Ajouter la possibilité de modifier les données stockées sur les sommets et les arrêtes.
- Ajouter la possibilité de modifier le graphe lui-même.

  Bonus : implanter un algorithme de plus court chemin dans ce graphe.

  Dans la mesure du possible, proposez plusieurs solution pour le modèle de données

Vous pouvez utiliser le graphe suivant :



[ ]:
```
fn main() {

}
main();
```

Solution 1

[ ]:
```
use std::rc::Rc;
use std::cell::RefCell;

struct Node {
    name: String,
    edges: Vec<Rc<RefCell<Edge>>>
}

impl Node {
    fn new(name: &str) -> Node {
        Node { name: name.to_string(), edges: Vec::new() }
    }
}

impl Drop for Node {
    fn drop(&mut self) {
        println!("Dropping node {}", self.name);
    }
}

struct Edge {
    from: Rc<RefCell<Node>>,
    to: Rc<RefCell<Node>>,
    value: i32
}

impl Edge {
    fn new(from: Rc<RefCell<Node>>, value: i32, to: Rc<RefCell<Node>>) ->␣
 ↪Rc<RefCell<Edge>> {
        let edge = Rc::new(RefCell::new(Edge {from: from.clone(), to: to.
 ↪clone(), value}));
        from.borrow_mut().edges.push(edge.clone());
        to.borrow_mut().edges.push(edge.clone());
        edge
    }
}

impl Drop for Edge {
    fn drop(&mut self) {
        println!("Dropping edge");
```

```
    }
}

#[derive(Default)]
struct Graph {
    nodes: Vec<Rc<RefCell<Node>>>,
    edges: Vec<Rc<RefCell<Edge>>>
}

impl Graph {
    fn add_node(&mut self, name: &str) -> Rc<RefCell<Node>> {
        let node = Rc::new(RefCell::new(Node::new(name)));
        self.nodes.push(node.clone());
        node
    }

    fn add_edge(&mut self, from: Rc<RefCell<Node>>, value: i32, to:␣
 ↪Rc<RefCell<Node>>) -> Rc<RefCell<Edge>> {
        let edge = Edge::new(from, value, to);
        self.edges.push(edge.clone());
        edge
    }
}

fn main() {
    let mut graph = Graph::default();
    let a = graph.add_node("a");
    let b = graph.add_node("b");
    let c = graph.add_node("c");
    let d = graph.add_node("d");
    let e = graph.add_node("e");
    let f = graph.add_node("f");
    let g = graph.add_node("g");
    let h = graph.add_node("h");
    let i = graph.add_node("i");

    graph.add_edge(a.clone(), 8, c.clone());
    graph.add_edge(a.clone(), 6, d.clone());
}
main();
```

Solution 1 bis

[ ]:
```
use std::rc::Rc;
use std::rc::Weak;
use std::cell::RefCell;
```

```rust
struct Node {
    name: String,
    edges: Vec<Rc<RefCell<Edge>>>
}

impl Node {
    fn new(name: &str) -> Node {
        Node { name: name.to_string(), edges: Vec::new() }
    }
}

impl Drop for Node {
    fn drop(&mut self) {
        println!("Dropping node {}", self.name);
    }
}

struct Edge {
    from: Weak<RefCell<Node>>,
    to: Weak<RefCell<Node>>,
    value: i32
}

impl Edge {
    fn new(from: Rc<RefCell<Node>>, value: i32, to: Rc<RefCell<Node>>) ->␣
 ↪Rc<RefCell<Edge>> {
        let edge = Rc::new(RefCell::new(Edge {from: Rc::downgrade(&from), to:␣
 ↪Rc::downgrade(&to), value}));
        from.borrow_mut().edges.push(edge.clone());
        to.borrow_mut().edges.push(edge.clone());
        edge
    }
}

impl Drop for Edge {
    fn drop(&mut self) {
        println!("Dropping edge");
    }
}

#[derive(Default)]
struct Graph {
    nodes: Vec<Rc<RefCell<Node>>>,
    edges: Vec<Rc<RefCell<Edge>>>
}

impl Graph {
```

```
    fn add_node(&mut self, name: &str) -> Rc<RefCell<Node>> {
        let node = Rc::new(RefCell::new(Node::new(name)));
        self.nodes.push(node.clone());
        node
    }

    fn add_edge(&mut self, from: Rc<RefCell<Node>>, value: i32, to:␣
 ↪Rc<RefCell<Node>>) -> Rc<RefCell<Edge>> {
        let edge = Edge::new(from, value, to);
        self.edges.push(edge.clone());
        edge
    }
}

fn main() {
    let mut graph = Graph::default();
    let a = graph.add_node("a");
    let b = graph.add_node("b");
    let c = graph.add_node("c");
    let d = graph.add_node("d");
    let e = graph.add_node("e");
    let f = graph.add_node("f");
    let g = graph.add_node("g");
    let h = graph.add_node("h");
    let i = graph.add_node("i");

    graph.add_edge(a.clone(), 8, c.clone());
    graph.add_edge(a.clone(), 6, d.clone());
}
main();
```

Solution 2

[ ]:
```
#[derive(Debug)]
struct Node {
    name: String,
    edges: Vec<Edge>
}

impl Node {
    fn new(name: &str) -> Node {
        Node { name: name.to_string(), edges: Vec::new() }
    }
}

#[derive(Debug)]
struct Edge {
```

```
    to: usize,
    value: i32
}

impl Edge {
    fn new(value: i32, to: usize) -> Edge {
        Edge {to, value}
    }
}

#[derive(Debug, Default)]
struct Graph {
    nodes: Vec<Node>
}

impl Graph {
    fn add_node(&mut self, name: &str) -> usize {
        let id = self.nodes.len();
        self.nodes.push(Node::new(name));
        id
    }

    fn add_edge(&mut self, from: usize, value: i32, to: usize) {
        self.nodes[from].edges.push(Edge::new(value, to));
        self.nodes[to].edges.push(Edge::new(value, from));
    }
}

fn main() {
    let mut graph = Graph::default();
    let a = graph.add_node("a");
    let b = graph.add_node("b");
    let c = graph.add_node("c");
    let d = graph.add_node("d");
    let e = graph.add_node("e");
    let f = graph.add_node("f");
    let g = graph.add_node("g");
    let h = graph.add_node("h");
    let i = graph.add_node("i");

    graph.add_edge(a, 8, c);
    graph.add_edge(a, 6, d);

    println!("{:#?}", graph);
}
main();
```