Dijkstra’s algorithm

• The idea: reminiscent of BFS, but adapted to handle weights
  – Grow the set of nodes whose shortest distance has been computed
  – Nodes not in the set will have a “best distance so far”
  – A priority queue will turn out to be useful for efficiency
Dijkstra’s Algorithm: Idea

- Initially, start node has cost 0 and all other nodes have cost $\infty$
- At each step:
  - Pick closest unknown vertex $v$
  - Add it to the “cloud” of known vertices
  - Update distances for nodes with edges from $v$
- That’s it! (But we need to prove it produces correct answers)
The Algorithm

1. For each node $v$, set $v.cost = \infty$ and $v.known = \text{false}$
2. Set $\text{source.cost} = 0$
3. While there are unknown nodes in the graph
   a) Select the unknown node $v$ with lowest cost
   b) Mark $v$ as known
   c) For each edge $(v,u)$ with weight $w$,
      \[
      c1 = v.cost + w \quad // \text{cost of best path through } v \text{ to } u \\
      c2 = u.cost \quad // \text{cost of best path to } u \text{ previously known} \\
      \text{if}(c1 < c2) \{ \quad // \text{if the path through } v \text{ is better} \\
          u.cost = c1 \\
          u.path = v \quad // \text{for computing actual paths} \\
      \}\]
Important features

• When a vertex is marked known, the cost of the shortest path to that node is known
  – The path is also known by following back-pointers

• While a vertex is still not known, another shorter path to it *might* still be found
Example #2

Order Added to Known Set:

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>??</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>??</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>??</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>??</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>??</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>??</td>
<td></td>
</tr>
</tbody>
</table>
Example #2

Order Added to Known Set:

A

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>??</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>≤ 2</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>≤ 1</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>??</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>??</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>??</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Order Added to Known Set:

A, D

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>≤ 6</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>≤ 2</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>≤ 2</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>≤ 7</td>
<td>D</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>≤ 6</td>
<td>D</td>
</tr>
</tbody>
</table>
Example #2

Order Added to Known Set:
A, D, C

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>≤ 6</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>≤ 2</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>≤ 4</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>≤ 6</td>
<td>D</td>
</tr>
</tbody>
</table>
Example #2

Order Added to Known Set:
A, D, C, E

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>≤ 3</td>
<td>E</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>Y</td>
<td>2</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>≤ 4</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>≤ 6</td>
<td>D</td>
</tr>
</tbody>
</table>
Example #3

How will the best-cost-so-far for Y proceed?

Is this expensive?
Example #3

How will the best-cost-so-far for Y proceed? 90, 81, 72, 63, 54, ...

Is this expensive? No, each edge is processed only once
A Greedy Algorithm

• Dijkstra’s algorithm
  – For single-source shortest paths in a weighted graph (directed or undirected) with no negative-weight edges

• An example of a *greedy algorithm*:
  – At each step, irrevocably does what seems best at that step
    • A locally optimal step, not necessarily globally optimal
  – Once a vertex is known, it is not revisited
    • Turns out to be globally optimal
Where are We?

• Had a problem: Compute shortest paths in a weighted graph with no negative weights

• Learned an algorithm: Dijkstra’s algorithm

• What should we do after learning an algorithm?
  – Prove it is correct
    • Not obvious!
    • We will sketch the key ideas
  – Analyze its efficiency
    • Will do better by using a data structure we learned earlier!
Correctness: Intuition

Rough intuition:

All the “known” vertices have the correct shortest path
  – True initially: shortest path to start node has cost 0
  – If it stays true every time we mark a node “known”, then by induction this holds and eventually everything is “known”

Key fact we need: When we mark a vertex “known” we won’t discover a shorter path later!
  – This holds only because Dijkstra’s algorithm picks the node with the next shortest path-so-far
  – The proof is by contradiction...
Correctness: The Cloud (Rough Sketch)

Suppose $v$ is the next node to be marked known ("added to the cloud")

- The **best-known path** to $v$ must have only nodes "in the cloud"
  - Else we would have picked a node closer to the cloud than $v$
- Suppose the **actual shortest path** to $v$ is different
  - It won’t use only cloud nodes, or we would know about it
  - So it must use non-cloud nodes. Let $w$ be the *first* non-cloud node on this path. The part of the path up to $w$ is already known and must be shorter than the best-known path to $v$. So $v$ would not have been picked. Contradiction.
Naïve asymptotic running time

• So far: $O(|V|^2)$

• We had a similar “problem” with topological sort being $O(|V|^2)$ due to each iteration looking for the node to process next
  – We solved it with a queue of zero-degree nodes
  – But here we need the lowest-cost node and costs can change as we process edges

• Solution?
Improving asymptotic running time

• So far: \( O(|V|^2) \)

• We had a similar “problem” with topological sort being \( O(|V|^2) \) due to each iteration looking for the node to process next
  – We solved it with a queue of zero-degree nodes
  – But here we need the lowest-cost node and costs can change as we process edges

• Solution?
  – A priority queue holding all unknown nodes, sorted by cost
  – But must support \texttt{decreaseKey} operation
    • Must maintain a reference from each node to its current position in the priority queue
    • Conceptually simple, but can be a pain to code up
Efficiency, second approach

Use pseudocode to determine asymptotic run-time

dijkstra(Graph G, Node start) {
    for each node: x.cost=infinity, x.known=false
    start.cost = 0
    build-heap with all nodes
    while(heap is not empty) {
        b = deleteMin()
        b.known = true
        for each edge (b,a) in G
            if(!a.known)
                if(b.cost + weight((b,a)) < a.cost){
                    decreaseKey(a,"new cost - old cost")
                    a.path = b
                }
    }
}
Efficiency, second approach

Use pseudocode to determine asymptotic run-time

```plaintext
dijkstra(Graph G, Node start) {
    for each node: x.cost=\infty, x.known=false
    start.cost = 0
    build-heap with all nodes
    while(heap is not empty) {
        b = deleteMin()
        b.known = true
        for each edge (b,a) in G
            if(!a.known)
                if(b.cost + weight((b,a)) < a.cost){
                    decreaseKey(a,"new cost - old cost")
                    a.path = b
                }
    }
}
```

Efficiency,

- \(O(|V|)\)
- \(O(|V|\log|V|)\)
- \(O(|E|\log|V|)\)
- \(O(|V|\log|V| + |E|\log|V|)\)
CSE373: Data Structures & Algorithms
Software-Design Interlude – Preserving Abstractions

Hunter Zahn
Summer 2016
Motivation

• Essential: knowing available data structures and their trade-offs
  – You’re taking a whole course on it! 😊

• However, you will rarely if ever re-implement these “in real life”
  – Provided by libraries

• But the key idea of an abstraction arises all the time “in real life”
  – Clients do not know how it is implemented
  – Clients do not need to know
  – Clients cannot “break the abstraction” no matter what they do
Interface vs. implementation

• Provide a reusable interface without revealing implementation

• More difficult than it sounds due to aliasing and field-assignment
  – Some common pitfalls

• So study it in terms of ADTs vs. data structures
  – Will use priority queues as example in lecture, but any ADT would do
  – Key aspect of grading your homework on graphs
Recall the abstraction

Clients:
“not trusted by ADT implementer”

• Can perform any sequence of ADT operations
• Can do anything type-checker allows on any accessible objects

Data structure:

new PQ(...)
insert(...)
deleteMin(...)
isEmpty()

• Should document how operations can be used and what is checked (raising appropriate exceptions)
  – E.g., parameter for method x not null

• If used correctly, correct priority queue for any client

• Client “cannot see” the implementation
  – E.g., binary min heap
Our example

• A priority queue with to-do items, so earlier dates “come first”
  – Simpler example than using Java generics

• Exact method names and behavior not essential to example

```java
public class Date {
    // some private fields (year, month, day)
    public int getYear() {...}
    public void setYear(int y) {...}
    // more methods
}
public class ToDoItem {
    // some private fields (date, description)
    public void setDate(Date d) {...}
    public void setDescription(String d) {...}
    // more methods
}
// continued next slide...
```
Our example

- A priority queue with to-do items, so earlier dates “come first”
  - Simpler example than using Java generics

- Exact method names and behavior not essential to example

```java
public class Date { … }
public class ToDoItem { … }
public class ToDoPQ {
    … // some private fields (array, size, …)
    public ToDoPQ() {…}
    void insert(ToDoItem t) {…}
    ToDoItem deleteMin() {…}
    boolean isEmpty() {…}
}
```
An obvious mistake

• Why we trained you to “mindlessly” make fields \texttt{private}:

```java
public class ToDoPQ {
    ... // other fields
    public ToDoItem[] heap;
    public ToDoPQ() {...}
    void insert(ToDoItem t) {...}
    ...
}

// client:
pq = new ToDoPQ();
pq.heap = null;
pq.insert(...); // likely exception
```

• Today’s lecture: \texttt{private} does not solve all your problems!
  – Upcoming pitfalls can occur even with all \texttt{private} fields
Less obvious mistakes

```java
public class ToDoPQ {
    ... // all private fields
    public ToDoPQ() {...}
    void insert(ToDoItem i) {...}
    ...
}

// client:
ToDoPQ pq = new ToDoPQ();
ToDoItem i = new ToDoItem(...);
pq.insert(i);
i.setDescription("some different thing");
pq.insert(i); // same object after update
x = deleteMin(); // x's description???
y = deleteMin(); // y's description???
```
Aliasing and mutation

- Client was able to update something inside the abstraction because client had an alias to it!
  - It is too hard to reason about and document what should happen, so better software designs avoid the issue!
More bad clients

```java
ToDoPQ pq = new ToDoPQ();
ToDoItem i1 = new ToDoItem(...); // year 2013
ToDoItem i2 = new ToDoItem(...); // year 2014
pq.insert(i1);
pq.insert(i2);
i1.setDate(...); // year 2015
x = deleteMin(); // “wrong” (???) item?
    // What date does returned item have???
```
More bad clients

i1

pq

heap:
size: 2
...

date:
description: “…”
year: ...
month: ...
...

date:
description: “…”
year: ...
month: ...
...

i2
More bad clients

```java
pq = new ToDoPQ();
ToDoItem i1 = new ToDoItem(...);
pq.insert(i1);
i1.setDate(null);
ToDoItem i2 = new ToDoItem(...);
pq.insert(i2); // NullPointerException???
```

Get exception inside data-structure code even if `insert` did a careful check that the date in the `ToDoItem` is not null

- Bad client later invalidates the check
The general fix

• Avoid aliases into the internal data (the “red arrows”) by copying objects as needed
  – Do not use the same objects inside and outside the abstraction because two sides do not know all mutation (field-setting) that might occur
  – “Copy-in-copy-out”

• A first attempt:

public class ToDoPQ {
    
    void insert(ToDoItem i) {
        ToDoItem internal_i =
            new ToDoItem(i.date, i.description);
        ... // use only the internal object
    }
}
Must copy the object

public class ToDoPQ {
    ...
    void insert(ToDoItem i) {
        ToDoItem internal_i =
            new ToDoItem(i.date,i.description);
        ... // use only the internal object
    }
}

• Notice this version accomplishes nothing
  – Still the alias to the object we got from the client:

public class ToDoPQ {
    ...
    void insert(ToDoItem i) {
        ToDoItem internal_i = i;
        ... // internal_i refers to same object
    }
}
Copying works...

```
ToDoItem i = new ToDoItem(...);
pq = new ToDoPQ();
pq.insert(i);
i.setDescription("some different thing");
pq.insert(i);
x = deleteMin();
y = deleteMin();
```
Didn’t do enough copying yet

Date d = new Date(…)
ToDoItem i = new ToDoItem(d,"buy beer");
pq = new ToDoPQ();
pq.insert(i);
d.setYear(2015);
...
Deep copying

- For copying to work fully, usually need to also make copies of all objects referred to (and that they refer to and so on...)
  - All the way down to `int`, `double`, `String`, ...
  - Called *deep copying* (versus our first attempt *shallow-copy*)

- Rule of thumb: Deep copy of things passed into abstraction

```java
public class ToDoPQ {
    ...
    void insert(ToDoItem i) {
        ToDoItem internal_i =
            new ToDoItem(new Date(...),
                         i.description);
        ...
        // use only the internal object
    }
}
```
Constructors take input too

- General rule: Do not “trust” data passed to constructors
  - Check properties and make deep copies

- Example: Floyd’s algorithm for `buildHeap` should:
  - Check the array (e.g., for `null` values in fields of objects or array positions)
  - Make a deep copy: new array, new objects

```java
public class ToDoPQ {
    // a second constructor that uses
    // Floyd’s algorithm, but good design
    // deep-copies the array (and its contents)
    void PriorityQueue(ToDoItem[] items) {
        ...
    }
}
```
That was copy-in, now copy-out...

• So we have seen:
  – Need to deep-copy data passed into abstractions to avoid pain and suffering

• Next:
  – Need to deep-copy data passed out of abstractions to avoid pain and suffering (unless data is “new” or no longer used in abstraction)

• Then:
  – If objects are immutable (no way to update fields or things they refer to), then copying unnecessary
**deleteMin is fine**

```java
public class ToDoPQ {
    ...
    ToDoItem deleteMin() {
        ToDoItem ans = heap[0];
        ...
        // algorithm involving percolateDown
        return ans;
    }
}
```

- Does not create a “red arrow” because object returned is no longer part of the data structure
- Returns an alias to object that was in the heap, but now it is not, so conceptual “ownership” “transfers” to the client
getMin needs copying

```
public class ToDoPQ {
    ToDoItem getMin() {
        int ans = heap[0];
        return ans;
    }
}
```

- Uh-oh, creates a “red arrow”

**ToDoItem**

```
ToDoItem i = new ToDoItem(...);
pq = new ToDoPQ();
x = pq.getMin();
x.setDate(...);
```
The fix

• Just like we deep-copy objects from clients before adding to our data structure, we should deep-copy parts of our data structure and return the copies to clients

• Copy-in and copy-out

```java
public class ToDoPQ {
    ToDoItem getMin() {
        int ans = heap[0];
        return new ToDoItem(new Date(...),
                        ans.description);
    }
}
```
Less copying

• (Deep) copying is one solution to our aliasing problems

• Another solution is **immutability**
  – Make it so nobody can ever change an object or any other objects it can refer to (deeply)
  – Allows “red arrows”, but immutability makes them harmless

• In Java, a **final** field cannot be updated after an object is constructed, so helps ensure immutability
  – But **final** is a “shallow” idea and we need “deep” immutability
This works

```java
public class Date {
    private final int year;
    private final String month;
    private final String day;
}
public class ToDoItem {
    private final Date date;
    private final String description;
}
public class ToDoPQ {
    void insert(ToDoItem i){/*no copy-in needed!*/}
    ToDoItem getMin(){/*no copy-out needed!*/
...
}
```

Notes:

- **String objects are immutable in Java**
- *(Using String for month and day is not great style though)*
This does not work

```java
public class Date {
    private final int year;
    private String month; // not final
    private final String day;
    ...
}
public class ToDoItem {
    private final Date date;
    private final String description;
}
public class ToDoPQ {
    void insert(ToDoItem i){/*no copy-in*/}
    ToDoItem getMin(){/*no copy-out*/}
    ...
}
```

Client could mutate a Date’s month that is in our data structure
- So must do entire deep copy of ToDoItem
**final is shallow**

```java
public class ToDoItem {
    private final Date date;
    private final String description;
}
```

- Here, `final` means no code can update the `date` or `description` fields after the object is constructed.
- So they will always refer to the same `Date` and `String` objects.
- But what if those objects have their contents change?
  - Cannot happen with `String` objects.
  - For `Date` objects, depends how we define `Date`.
- So `final` is a “shallow” notion, but we can use it “all the way down” to get deep immutability.
This works

- When deep-copying, can “stop” when you get to immutable data
  - Copying immutable data is wasted work, so poor style

```java
public class Date {
    // immutable
    private final int year;
    private final String month;
    private final String day;
    ...
}

public class ToDoItem {
    private Date date;
    private String description;
}

public class ToDoPQ {
    ToDoItem getMin() {
        int ans = heap[0];
        return new ToDoItem(ans.date, // okay!
                            ans.description);
    }
}
```
What about this?

```java
public class Date { // immutable
    ...
}
public class ToDoItem { // immutable (unlike last slide)
    ...
}
public class ToDoPQ {
    // a second constructor that uses
    // Floyd’s algorithm
    void PriorityQueue(ToDoItem[] items) {
        // what copying should we do?
        ...
    }
}
```
What about this?

```java
public class Date { // immutable
    ...
}
public class ToDoItem { // immutable (unlike last slide)
    ...
}
public class ToDoPQ {
    // a second constructor that uses
    // Floyd’s algorithm
    void PriorityQueue(ToDoItem[] items) {
        // what copying should we do?
        ...
    }
}
```

Copy the array, but do not copy the `ToDoItem` or `Date` objects
Homework 4

• You are implementing a graph abstraction

• As provided, `Vertex` and `Edge` are immutable
  – But `Collection<Vertex>` and `Collection<Edge>` are not

• You might choose to add fields to `Vertex` or `Edge` that make them not immutable
  – Leads to more copy-in-copy-out, but that’s fine!

• *Or* you might leave them immutable and keep things like “best-path-cost-so-far” in another dictionary (e.g., a `HashMap`)

*There is more than one good design, but preserve your abstraction*
  – *Great practice with a key concept in software design*