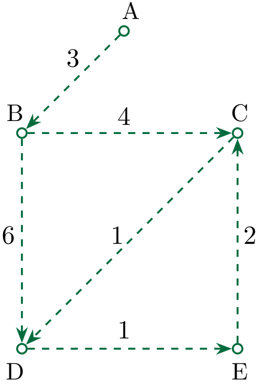
Functional Correctness of Dijkstra's, Kruskal's, and Prim's Algorithms in C

Anshuman Mohan   Leow Wei Xiang   Aquinas Hobor

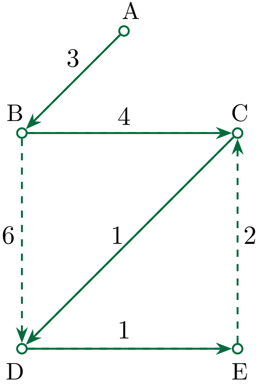
National University of Singapore

CAV 2021
July 18-24, 2021



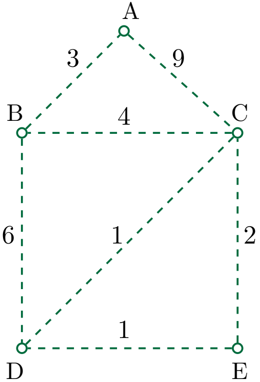
Dijkstra's algorithm (1959)
one-to-all shortest paths

Refresher: Dijkstra, Prim, Kruskal



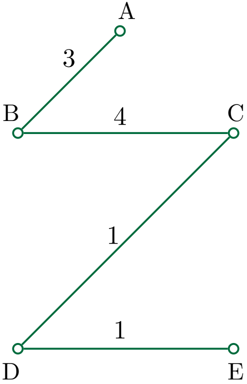
Dijkstra's algorithm (1959)
one-to-all shortest paths

Refresher: Dijkstra, Prim, Kruskal



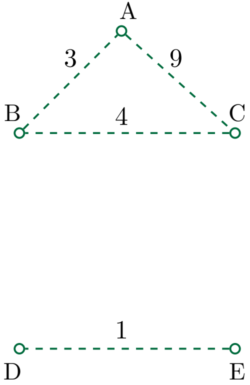
(Jarník's) Prim's (Dijkstra's) algorithm (1930, 1957, 1959)
prune connected graph to MST

Refresher: Dijkstra, Prim, Kruskal



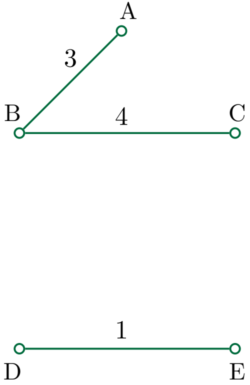
(Jarník's) Prim's (Dijkstra's) algorithm (1930, 1957, 1959)
prune connected graph to MST

Refresher: Dijkstra, Prim, Kruskal



Kruskal's algorithm (1956)
prune graph to MSF

Refresher: Dijkstra, Prim, Kruskal



Kruskal's algorithm (1956)
prune graph to MSF

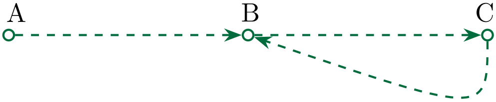
Motivation: a precondition for Dijkstra

In a graph with `size` vertices,
the longest possible optimal path has `size-1` links

Motivation: a precondition for Dijkstra

In a graph with `size` vertices,
the longest possible optimal path has `size-1` links
so edge costs should be $\leqslant \lfloor \texttt{MAX/(size-1)} \rfloor$ to prevent overflow

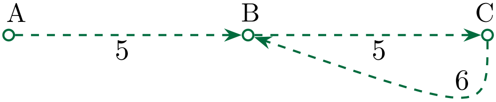
In a graph with `size` vertices,
the longest possible optimal path has `size-1` links
so edge costs should be $\leqslant \lfloor \texttt{MAX/(size-1)} \rfloor$ to prevent overflow



Motivation: a precondition for Dijkstra

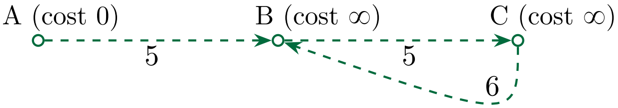
In a graph with `size` vertices,
the longest possible optimal path has `size-1` links
so edge costs should be $\leqslant \lfloor$`MAX/(size-1)`$\rfloor$ to prevent overflow

Consider a 4-bit machine and unsigned integers
`MAX` $= 15$, `size` $= 3$, so every edge-cost $\leqslant 7$.



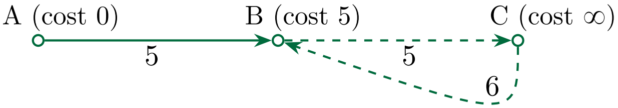
In a graph with `size` vertices,
the longest possible optimal path has `size-1` links
so edge costs should be $\leqslant \lfloor$`MAX/(size-1)`$\rfloor$ to prevent overflow

Consider a 4-bit machine and unsigned integers
`MAX` $= 15$, `size` $= 3$, so every edge-cost $\leqslant 7$.



In a graph with `size` vertices,
the longest possible optimal path has `size-1` links
so edge costs should be $\leqslant \lfloor \texttt{MAX/(size-1)} \rfloor$ to prevent overflow

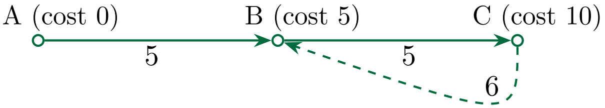
Consider a 4-bit machine and unsigned integers
$\texttt{MAX} = 15$, $\texttt{size} = 3$, so every edge-cost $\leqslant 7$.



Motivation: a precondition for Dijkstra

In a graph with `size` vertices,
the longest possible optimal path has `size-1` links
so edge costs should be $\leqslant \lfloor$`MAX/(size-1)`$\rfloor$ to prevent overflow

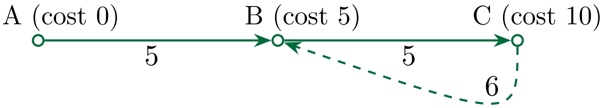
Consider a 4-bit machine and unsigned integers
`MAX` $= 15$, `size` $= 3$, so every edge-cost $\leqslant 7$.



Motivation: a precondition for Dijkstra

In a graph with `size` vertices,
the longest possible optimal path has `size-1` links
so edge costs should be $\leqslant \lfloor \texttt{MAX/(size-1)} \rfloor$ to prevent overflow

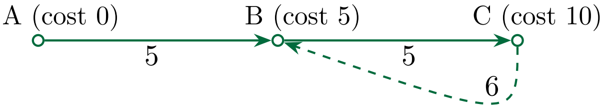
Consider a 4-bit machine and unsigned integers
$\texttt{MAX} = 15$, $\texttt{size} = 3$, so every edge-cost $\leqslant 7$.



if $5 > 16$ then relax C $\rightsquigarrow$ B

In a graph with `size` vertices,
the longest possible optimal path has `size-1` links
so edge costs should be $\leqslant \lfloor \texttt{MAX/(size-1)} \rfloor$ to prevent overflow

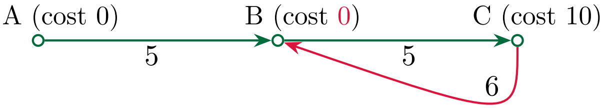
Consider a 4-bit machine and unsigned integers
$\texttt{MAX} = 15$, $\texttt{size} = 3$, so every edge-cost $\leqslant 7$.



if $5 > 0$ then relax C $\rightsquigarrow$ B

In a graph with `size` vertices,
the longest possible optimal path has `size-1` links
so edge costs should be $\leqslant \lfloor \texttt{MAX/(size-1)} \rfloor$ to prevent overflow

Consider a 4-bit machine and unsigned integers
`MAX` $= 15$, `size` $= 3$, so every edge-cost $\leqslant 7$.



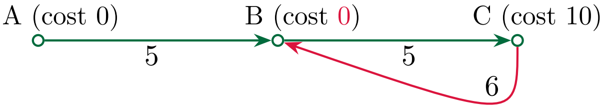
if $5 > 0$ then relax C $\rightsquigarrow$ B

Must allow room for the probing edge

Motivation: a precondition for Dijkstra

In a graph with **`size`** vertices,
the longest possible optimal path has **`size-1`** links
so edge costs should be $\leqslant \lfloor$`MAX/(size-1)`$\rfloor$ to prevent overflow

Consider a 4-bit machine and unsigned integers
`MAX` $= 15$, **`size`** $= 3$, so every edge-cost $\leqslant 7$.



if $5 > 0$ then relax C $\rightsquigarrow$ B

Must allow room for the probing edge
so an edge-cost is, at most, $\lfloor$`MAX/size`$\rfloor$

There are many ways to fix this!

Motivation: A precondition for Dijkstra

There are many ways to fix this!

Refactor troublesome addition as subtraction

Motivation: A precondition for Dijkstra

There are many ways to fix this!

- Refactor troublesome addition as subtraction
- Coerce to `long`

Motivation: A precondition for Dijkstra

There are many ways to fix this!

- Refactor troublesome addition as subtraction
- Coerce to `long`
- Work in `float`, which has ∞^+

Motivation: A precondition for Dijkstra

There are many ways to fix this!

- Refactor troublesome addition as subtraction
- Coerce to `long`
- Work in `float`, which has ∞^+
- Never look back into optimized part

There are many ways to fix this!

- Refactor troublesome addition as subtraction
- Coerce to `long`
- Work in `float`, which has ∞^+
- Never look back into optimized part
- Stop earlier: when you have one vertex left in PQ, rather than zero

There are many ways to fix this!

- Refactor troublesome addition as subtraction
- Coerce to `long`
- Work in `float`, which has ∞^+
- Never look back into optimized part
- Stop earlier: when you have one vertex left in PQ, rather than zero

Sadly, this is code directly from textbooks, and intuition supports our misstep

Motivation: A precondition for Dijkstra

There are many ways to fix this!

- Refactor troublesome addition as subtraction
- Coerce to `long`
- Work in `float`, which has ∞^+
- Never look back into optimized part
- Stop earlier: when you have one vertex left in PQ, rather than zero

Sadly, this is code directly from textbooks, and
intuition supports our misstep...
...bugs such as this are often overlooked

CertiGraph: problem scope

Verified
Software
Toolchain

Certifying Graph-Manipulating C Programs via Localizations within Data Structures

SHENGYI WANG, National University of Singapore, Singapore
QINXIANG CAO, Shanghai Jiao Tong University, China
ANSHUMAN MOHAN, National University of Singapore, Singapore
AQUINAS HOBOR, National University of Singapore, Singapore

CompCert + VST + <u>CertiGraph</u>

CertiGraph: problem scope

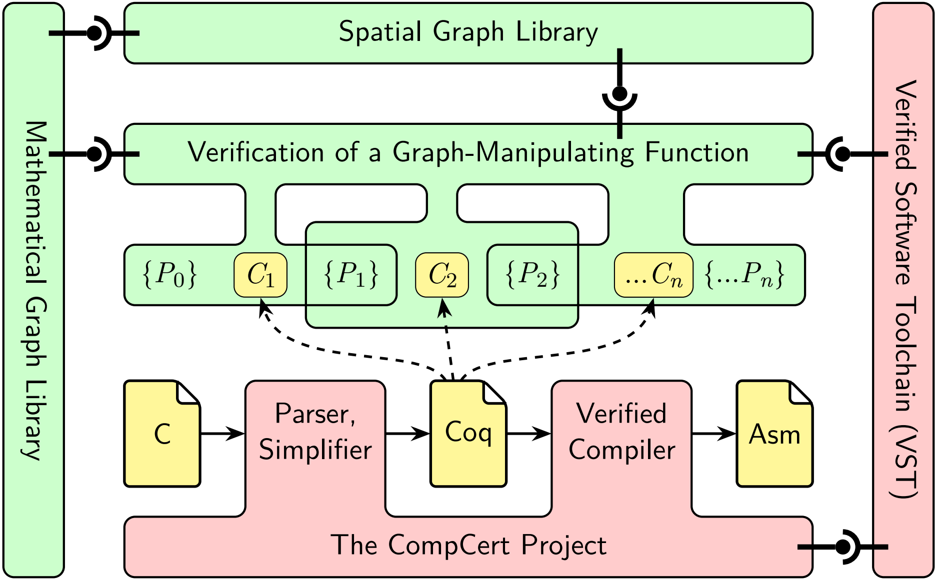
Verified
Software
Toolchain

Certifying Graph-Manipulating C Programs via Localizations within Data Structures

SHENGYI WANG, National University of Singapore, Singapore
QINXIANG CAO, Shanghai Jiao Tong University, China
ANSHUMAN MOHAN, National University of Singapore, Singapore
AQUINAS HOBOR, National University of Singapore, Singapore

CompCert + VST + <u>CertiGraph</u>

Verify executable graph-manipulating code with rich specifications


Spatial Graph Library
Verification of a Graph-Manipulating Function
$\{P_0\}$
C_1
$\{P_1\}$
C_2
$\{P_2\}$
$\ldots C_n$
$\{\ldots P_n\}$
Mathematical Graph Library
Verified Software Toolchain (VST)
C
Parser, Simplifier
Coq
Verified Compiler
Asm
The CompCert Project

Outline

1. CertiGraph: Motivation and Overview
2. Mathematical and Spatial Representations
3. Shortest Path: Dijkstra
4. Minimum Spanning Forest: Prim and Kruskal

Outline

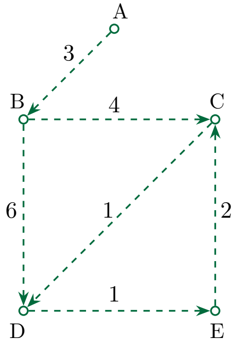
1. CertiGraph: Motivation and Overview

2. Mathematical and Spatial Representations

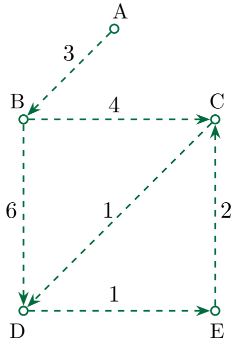
3. Shortest Path: Dijkstra

4. Minimum Spanning Forest: Prim and Kruskal

Supporting edge-labeled adjacency matrices

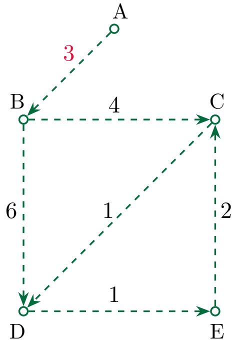


Supporting edge-labeled adjacency matrices



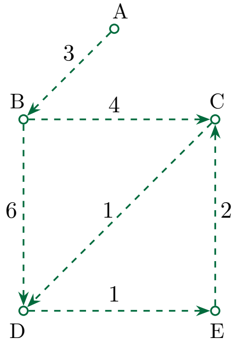
	A	B	C	D	E
A	∞	3	∞	∞	∞
B	∞	∞	4	6	∞
C	∞	∞	∞	1	∞
D	∞	∞	∞	∞	1
E	∞	∞	2	∞	∞

Supporting edge-labeled adjacency matrices



	A	B	C	D	E
A	∞	3	∞	∞	∞
B	∞	∞	4	6	∞
C	∞	∞	∞	1	∞
D	∞	∞	∞	∞	1
E	∞	∞	2	∞	∞

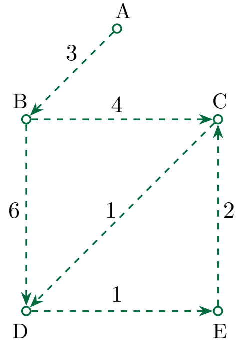
Supporting edge-labeled adjacency matrices



	A	B	C	D	E
A	∞	3	∞	∞	∞
B	∞	∞	4	6	∞
C	∞	∞	∞	1	∞
D	∞	∞	∞	∞	1
E	∞	∞	2	∞	∞

Requirement 1: graph, not multigraph

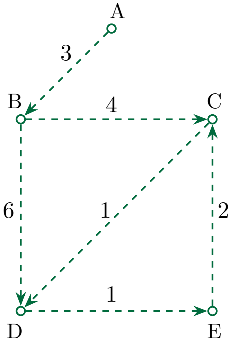
Supporting edge-labeled adjacency matrices



	A	B	C	D	E
A	∞	3	∞	∞	∞
B	∞	∞	4	6	∞
C	∞	∞	∞	1	∞
D	∞	∞	∞	∞	1
E	∞	∞	2	∞	∞

Requirement 1: graph, not multigraph
Requirement 2: labels representable

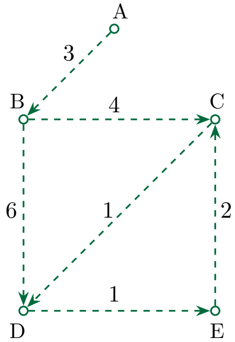
Supporting edge-labeled adjacency matrices



	A	B	C	D	E
A	∞	3	∞	∞	∞
B	∞	∞	4	6	∞
C	∞	∞	∞	1	∞
D	∞	∞	∞	∞	1
E	∞	∞	2	∞	∞

Requirement 1: graph, not multigraph
Requirement 2: labels representable
Requirement 3: $\exists\infty.\ \infty$ representable

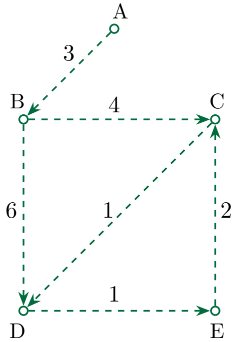
Supporting edge-labeled adjacency matrices



	A	B	C	D	E
A	∞	3	∞	∞	∞
B	∞	∞	4	6	∞
C	∞	∞	∞	1	∞
D	∞	∞	∞	∞	1
E	∞	∞	2	∞	∞

Requirement 1: graph, not multigraph
Requirement 2: labels representable
Requirement 3: $\exists\infty.\ \infty$ representable and no bona-fide edge has cost ∞

Supporting edge-labeled adjacency matrices

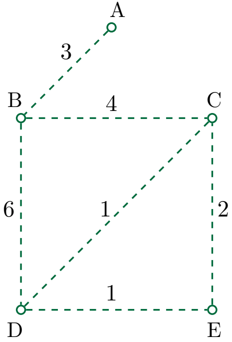


	A	B	C	D	E
A	∞	3	∞	∞	∞
B	∞	∞	4	6	∞
C	∞	∞	∞	1	∞
D	∞	∞	∞	∞	1
E	∞	∞	2	∞	∞

Requirement 1: graph, not multigraph
Requirement 2: labels representable
Requirement 3: $\exists \infty$. ∞ representable and no bona-fide edge has cost ∞

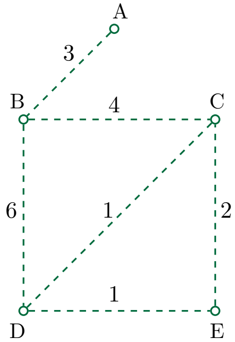
Contribution 1: integrate this notion of graphs into CertiGraph in a generic way

Undirected graphs



Kruskal and Prim handle undirected graphs

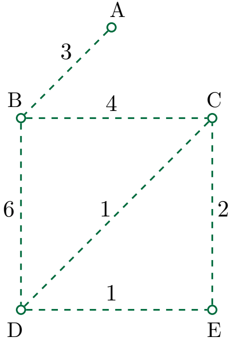
Undirected graphs



	A	B	C	D	E
A	∞	3	∞	∞	∞
B	∞	∞	4	6	∞
C	∞	∞	∞	1	2
D	∞	∞	∞	∞	1
E	∞	∞	∞	∞	∞

Kruskal and Prim handle undirected graphs

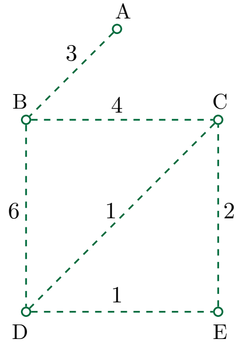
Undirected graphs



	A	B	C	D	E
A	∞	3	∞	∞	∞
B	∞	∞	4	6	∞
C	∞	∞	∞	1	2
D	∞	∞	∞	∞	1
E	∞	∞	∞	∞	∞

Kruskal and Prim handle undirected graphs

Contribution 2: integrate undirected graphs into CertiGraph



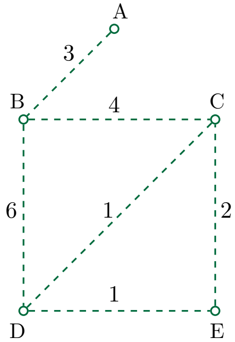
	A	B	C	D	E
A	∞	3	∞	∞	∞
B	∞	∞	4	6	∞
C	∞	∞	∞	1	2
D	∞	∞	∞	∞	1
E	∞	∞	∞	∞	∞

Kruskal and Prim handle undirected graphs

Contribution 2: integrate undirected graphs into CertiGraph

Build lightweight undirected definitions

Undirected graphs

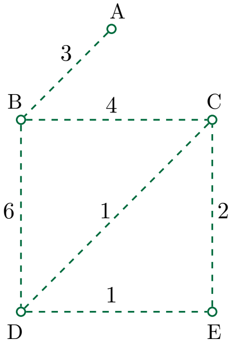


	A	B	C	D	E
A	∞	3	∞	∞	∞
B	∞	∞	4	6	∞
C	∞	∞	∞	1	2
D	∞	∞	∞	∞	1
E	∞	∞	∞	∞	∞

Kruskal and Prim handle undirected graphs

Contribution 2: integrate undirected graphs into CertiGraph

Build lightweight undirected definitions
Prove connections to existing directed definitions



	A	B	C	D	E
A	∞	3	∞	∞	∞
B	∞	∞	4	6	∞
C	∞	∞	∞	1	2
D	∞	∞	∞	∞	1
E	∞	∞	∞	∞	∞

Kruskal and Prim handle undirected graphs

Contribution 2: integrate undirected graphs into CertiGraph

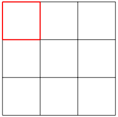
Build lightweight undirected definitions
Prove connections to existing directed definitions
Grow undirected infrastructure

Laying out graphs in memory

We support four representations of adjacency matrices in memory:

We support four representations of adjacency matrices in memory:

stack-allocated 2D array `int graph[size][size]`

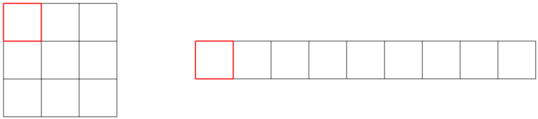


Laying out graphs in memory

We support four representations of adjacency matrices in memory:

stack-allocated 2D array `int graph[size][size]`

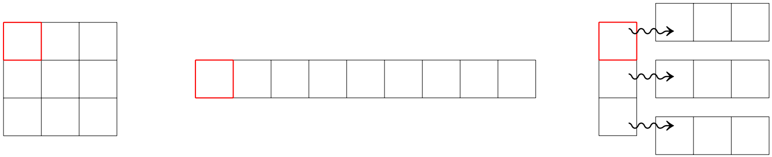
stack-allocated 1D array `int graph[size×size]`



Laying out graphs in memory

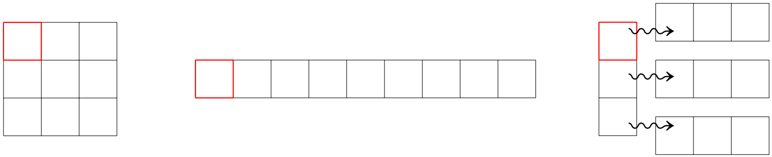
We support four representations of adjacency matrices in memory:

- stack-allocated 2D array `int graph[size][size]`
- stack-allocated 1D array `int graph[size×size]`
- heap-allocated 2D array `int **graph`



We support four representations of adjacency matrices in memory:

- stack-allocated 2D array `int graph[size][size]`
- stack-allocated 1D array `int graph[size×size]`
- heap-allocated 2D array `int **graph`

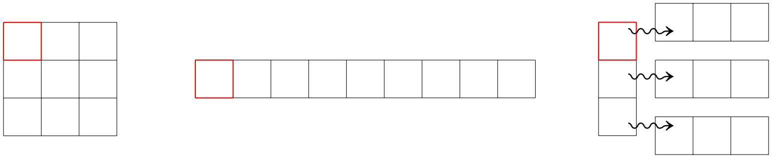


Contribution 3: separation logic for each into CertiGraph

Laying out graphs in memory

We support four representations of adjacency matrices in memory:

- stack-allocated 2D array `int graph[size][size]`
- stack-allocated 1D array `int graph[size×size]`
- heap-allocated 2D array `int **graph`



Contribution 3: separation logic for each into CertiGraph

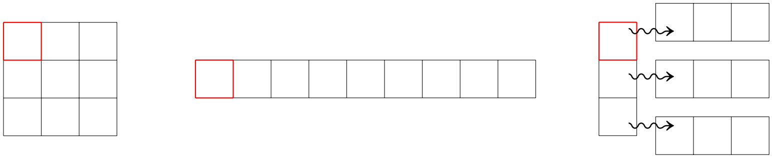
Well engineered: can swap the model with only minimal changes ($< 1\%$) to the formal proofs.

We support four representations of adjacency matrices in memory:

stack-allocated 2D array `int graph[size][size]`

stack-allocated 1D array `int graph[size×size]`

heap-allocated 2D array `int **graph`



Contribution 3: separation logic for each into CertiGraph

Well engineered: can swap the model with only minimal changes ($< 1\%$) to the formal proofs.

Contribution 3.1: separation logic for edge lists too

Outline

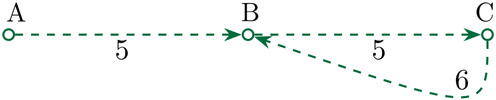
1. CertiGraph: Motivation and Overview

2. Mathematical and Spatial Representations

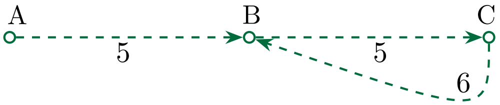
3. Shortest Path: Dijkstra

4. Minimum Spanning Forest: Prim and Kruskal

```
Class SoundDijk size inf g := {
  sadjmat: SoundAdjMat size inf g;
  efr: ∀e, evalid g e →
           0 ⩽ elabel g e ⩽ (MAX/size);
  ifr: (MAX/size) * (size-1) < inf;
  sz1: size = 1 → ∀e, evalid g e → elabel g e < inf
}.
```

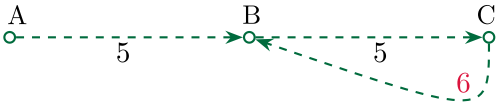


```
Class SoundDijk size inf g := {
  sadjmat: SoundAdjMat size inf g;
  efr: ∀e, evalid g e →
          0 ⩽ elabel g e ⩽ (MAX/size);
  ifr: (MAX/size) * (size-1) < inf;
  sz1: size = 1 → ∀e, evalid g e → elabel g e < inf
}.
```



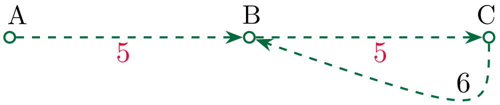
sadjmat: SoundDijk is an adjacency matrix

```
Class SoundDijk size inf g := {
  sadjmat: SoundAdjMat size inf g;
  efr: ∀e, evalid g e →
           0 ⩽ elabel g e ⩽ (MAX/size);
  ifr: (MAX/size) * (size-1) < inf;
  sz1: size = 1 → ∀e, evalid g e → elabel g e < inf
}.
```



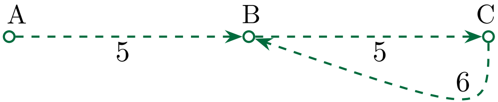
sadjmat: SoundDijk is an adjacency matrix
efr: Leave room for probing link

```
Class SoundDijk size inf g := {
  sadjmat: SoundAdjMat size inf g;
  efr: ∀e, evalid g e →
           0 ⩽ elabel g e ⩽ (MAX/size);
  ifr: (MAX/size) * (size-1) < inf;
  sz1: size = 1 → ∀e, evalid g e → elabel g e < inf
}.
```



`sadjmat:`	`SoundDijk` is an adjacency matrix
`efr:`	Leave room for probing link
`ifr:`	Bona-fide costs must dodge `inf`

```
Class SoundDijk size inf g := {
  sadjmat: SoundAdjMat size inf g;
  efr: ∀e, evalid g e →
           0 ⩽ elabel g e ⩽ (MAX/size);
  ifr: (MAX/size) * (size-1) < inf;
  sz1: size = 1 → ∀e, evalid g e → elabel g e < inf
}.
```



`sadjmat:`	`SoundDijk` is an adjacency matrix
`efr:`	Leave room for probing link
`ifr:`	Bona-fide costs must dodge `inf`
`sz1:`	Special bounds for degenerate case for `inf`

Dijkstra: code and specification

```
void dijkstra (int **g, int src, int *dist,
               int *prev, int size, int inf) {
 /* elided: init PQ, fill out dist and prev */
 while (size(pq)) {
```

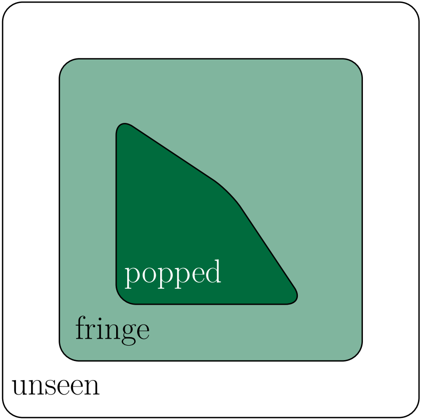
Dijkstra: code and specification

```
void dijkstra (int **g, int src, int *dist,
               int *prev, int size, int inf) {
 /* elided: init PQ, fill out dist and prev */
 while (size(pq)) {
```

$\{\exists dist, prev, popped.\ dijk\_correct(\gamma, \texttt{src}, popped, prev, dist)\}$

Dijkstra: intuition for *dijk_correct*

popped $\triangleq$ globally optimal path known
fringe $\triangleq$ locally optimal path known: popped parent + one hop
unseen $\triangleq$ no path exists from popped parent + one hop



Dijkstra: code and specification

```
while (size(pq)) {
```

$\{dijk\_correct(\gamma, \texttt{src}, popped, prev, dist)\}$

```
while (size(pq)) {
```

$\{dijk\_correct(\gamma, \texttt{src}, popped, prev, dist)\}$

```
  u = popMin(pq);
```

Dijkstra: code and specification

```
while (size(pq)) {
```

$\{dijk\_correct(\gamma, \texttt{src}, popped, prev, dist)\}$

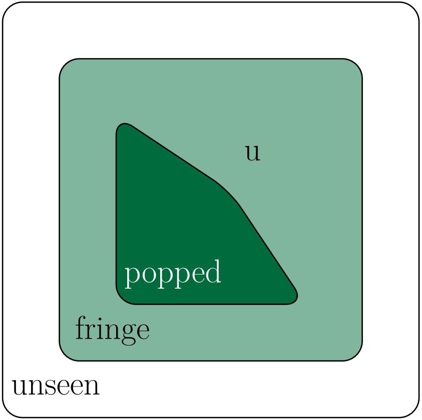
```
  u = popMin(pq);

  for (i = 0; i < size; i++) {
```

$\{\exists dist', prev'\ \ dijk\_correct\_weak(\gamma, \texttt{src}, popped \uplus \{\texttt{u}\}, prev', dist', \texttt{i}, \texttt{u})\}$

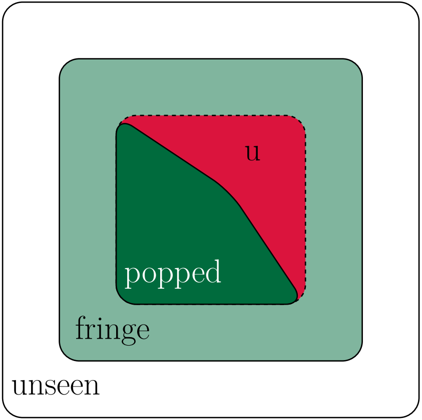
Dijkstra: intuition for *dijk_correct_weak*

popped $\triangleq$ globally optimal path known
fringe $\triangleq$ locally optimal path known: popped parent + one hop
unseen $\triangleq$ no path exists from popped parent + one hop
u $\triangleq$ cheapest in the fringe



Dijkstra: intuition for *dijk_correct_weak*

popped $\triangleq$ globally optimal path known
fringe $\triangleq$ locally optimal path known: popped parent + one hop
unseen $\triangleq$ no path exists from popped parent + one hop
u $\triangleq$ cheapest in the fringe



Dijkstra: code and specification

```
while (size(pq)) {
```

$\{dijk\_correct(\gamma, \texttt{src}, popped, prev, dist)\}$

```
  u = popMin(pq);
  for (i = 0; i < size; i++) {
```

$\{\exists dist', prev'.\ dijk\_correct\_weak(\gamma, \texttt{src}, popped \uplus \{\texttt{u}\}, prev', dist', \texttt{i}, \texttt{u})\}$

```
while (size(pq)) {
```

$\{ \mathit{dijk\_correct}(\gamma, \texttt{src}, \mathit{popped}, \mathit{prev}, \mathit{dist}) \}$

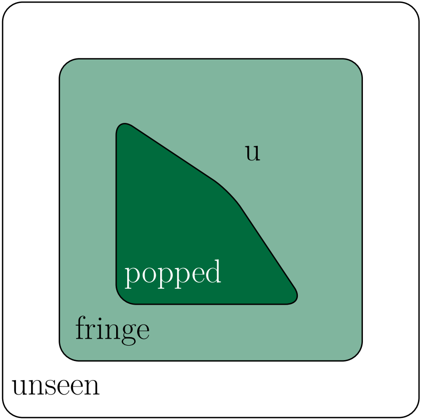
```
  u = popMin(pq);
  for (i = 0; i < size; i++) {
```

$\{ \exists \mathit{dist}', \mathit{prev}'.\ \mathit{dijk\_correct\_weak}(\gamma, \texttt{src}, \mathit{popped} \uplus \{\texttt{u}\}, \mathit{prev}', \mathit{dist}', \texttt{i}, \texttt{u}) \}$

```
  /* elided: potentially relax edge (u,i) */
}} /* for */
```

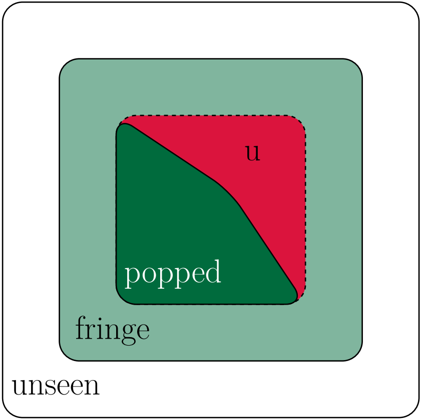
Dijkstra: intuition for recovering *dijk_correct*

popped $\triangleq$ globally optimal path known
fringe $\triangleq$ locally optimal path known:
popped parent + one hop
unseen $\triangleq$ no path exists from
popped parent + one hop
u $\triangleq$ cheapest in the fringe



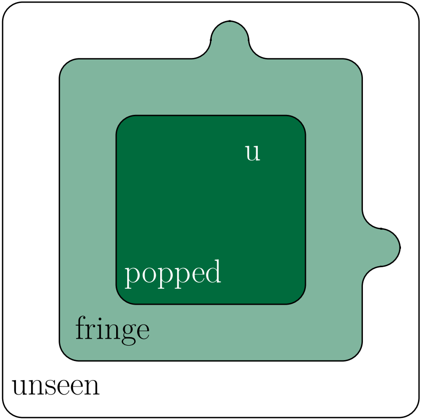
Dijkstra: intuition for recovering *dijk_correct*

popped $\triangleq$ globally optimal path known
fringe $\triangleq$ locally optimal path known: popped parent + one hop
unseen $\triangleq$ no path exists from popped parent + one hop
u $\triangleq$ cheapest in the fringe



Dijkstra: intuition for recovering *dijk_correct*

popped $\triangleq$ globally optimal path known
fringe $\triangleq$ locally optimal path known: popped parent + one hop
unseen $\triangleq$ no path exists from popped parent + one hop
u $\triangleq$ cheapest in the fringe



Dijkstra: code and specification

```
while (size(pq)) {
```

$\{dijk\_correct(\gamma, \texttt{src}, popped, prev, dist)\}$

```
  u = popMin(pq);
  for (i = 0; i < size; i++) {
```

$\{\exists dist', prev'.\ dijk\_correct\_weak(\gamma, \texttt{src}, popped \uplus \{\texttt{u}\}, prev', dist', \texttt{i}, \texttt{u})\}$

```
  /* elided: potentially relax edge (u,i) */
}} /* for */
```

Dijkstra: code and specification

```
while (size(pq)) {
```

$\{dijk\_correct(\gamma, \texttt{src}, popped, prev, dist)\}$

```
  u = popMin(pq);
  for (i = 0; i < size; i++) {
```

$\{\exists dist', prev'.\ dijk\_correct\_weak(\gamma, \texttt{src}, popped \uplus \{\texttt{u}\}, prev', dist', \texttt{i}, \texttt{u})\}$

```
  /* elided: potentially relax edge (u,i) */
}} /* for */    } /* while */
```

```
while (size(pq)) {
```

$\{dijk\_correct(\gamma, \mathtt{src}, popped, prev, dist)\}$

```
  u = popMin(pq);
  for (i = 0; i < size; i++) {
```

$\{\exists dist', prev'.\ dijk\_correct\_weak(\gamma, \mathtt{src}, popped \uplus \{\mathtt{u}\}, prev', dist', \mathtt{i}, \mathtt{u})\}$

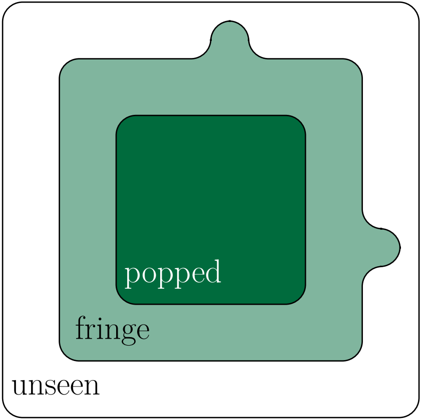
```
  /* elided: potentially relax edge (u,i) */
}} /* for */    } /* while */
```

$$\left\{\begin{array}{l}\exists dist'', prev''.\ \forall dst.\ 0 \leqslant dst < \mathtt{size} \rightarrow \\ \quad inv\_popped(\gamma, src, \gamma.V, prev'', dist'', dst)\end{array}\right\}$$

```
freePQ (pq); return;  } /* func */
```

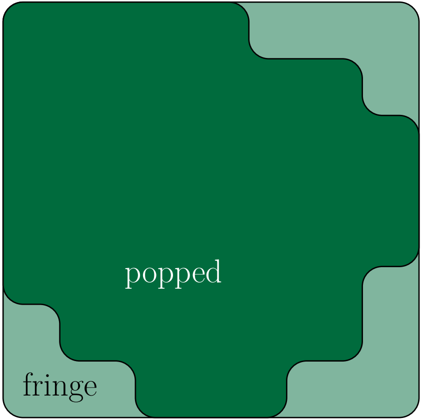
Dijkstra: postcondition

popped $\triangleq$ globally optimal path known
fringe $\triangleq$ locally optimal path known:
popped parent + one hop
unseen $\triangleq$ no path exists from
popped parent + one hop



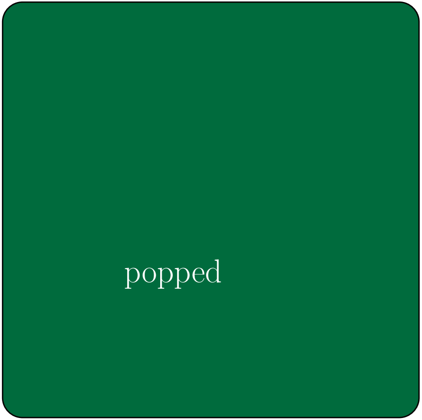
Dijkstra: postcondition

popped $\triangleq$ globally optimal path known
fringe $\triangleq$ locally optimal path known: popped parent + one hop
unseen $\triangleq$ no path exists from popped parent + one hop



Dijkstra: postcondition

popped $\triangleq$ globally optimal path known
fringe $\triangleq$ locally optimal path known:
popped parent + one hop
unseen $\triangleq$ no path exists from
popped parent + one hop



Contribution 4: machine-certified "real C" Dijkstra

Dijkstra: contributions

Contribution 4: machine-certified "real C" Dijkstra

Contribution 5: precise edge bounds to avoid overflow

Contribution 4: machine-certified "real C" Dijkstra

Contribution 5: precise edge bounds to avoid overflow

Contribution 6a: three adjacency matrix representations

Contribution 4: machine-certified "real C" Dijkstra

Contribution 5: precise edge bounds to avoid overflow

Contribution 6a: three adjacency matrix representations
(code bases differ by less than 1%)

Contribution 4: machine-certified "real C" Dijkstra

Contribution 5: precise edge bounds to avoid overflow

Contribution 6a: three adjacency matrix representations
(code bases differ by less than 1%)

Contribution 7: certified binary heap

Contribution 4: machine-certified "real C" Dijkstra

Contribution 5: precise edge bounds to avoid overflow

Contribution 6a: three adjacency matrix representations
(code bases differ by less than 1%)

Contribution 7: certified binary heap with `decrease-key`

Contribution 4: machine-certified "real C" Dijkstra

Contribution 5: precise edge bounds to avoid overflow

Contribution 6a: three adjacency matrix representations
(code bases differ by less than 1%)

Contribution 7: certified binary heap with `decrease-key`
(several subtle C over/underflows discovered)

```
#define ROOT_IDX 0
#define PARENT(x) (x - 1) / 2

void swim(unsigned int k, Item arr[],
          unsigned int lookup[])
{
 while (k > ROOT_IDX &&
        less (k, PARENT(k), arr)) {
   exch(k, PARENT(k), arr, lookup);
   k = PARENT(k);
 }
}
```

Over/underflows in binary heaps

```
#define ROOT_IDX 0
#define PARENT(x) (x - 1) / 2

void swim(unsigned int k, Item arr[],
          unsigned int lookup[])
{
 while (k > ROOT_IDX &&
        less (k, PARENT(k), arr)) {
   exch(k, PARENT(k), arr, lookup);
   k = PARENT(k);
 }
}
```

Over/underflows in binary heaps

```
#define ROOT_IDX 0
#define PARENT(x) (x - 1) / 2

void swim(unsigned int k, Item arr[],
          unsigned int lookup[])
{
 while (k > ROOT_IDX &&
        less (k, PARENT(k), arr)) {
   exch(k, PARENT(k), arr, lookup);
   k = PARENT(k);
 }
}
```

Over/underflows in binary heaps

```
#define ROOT_IDX 0
#define PARENT(x) (x - 1) / 2

void swim(unsigned int k, Item arr[],
          unsigned int lookup[])
{
 while (k > ROOT_IDX &&
        less (k, PARENT(k), arr)) {
   exch(k, PARENT(k), arr, lookup);
   k = PARENT(k);
 }
}
```

Over/underflows in binary heaps

```
#define ROOT_IDX 0
#define PARENT(x) (x - 1) / 2

void swim(unsigned int k, Item arr[],
          unsigned int lookup[])
{
 while (k > ROOT_IDX &&
        less (k, PARENT(k), arr)) {
   exch(k, PARENT(k), arr, lookup);
   k = PARENT(k);
 }
}
```

```
#define ROOT_IDX 0
#define PARENT(x) (x - 1) / 2

void swim(unsigned int k, Item arr[],
          unsigned int lookup[])
{
 while (k > ROOT_IDX &&
        less (k, PARENT(k), arr)) {
   exch(k, PARENT(k), arr, lookup);
   k = PARENT(k);
 }
}
```

Over/underflows in binary heaps

```
#define ROOT_IDX 0u
#define PARENT(x) (x - 1u) / 2u

void swim(unsigned int k, Item arr[],
          unsigned int lookup[])
{
 while (k > ROOT_IDX &&
       less (k, PARENT(k), arr)) {
   exch(k, PARENT(k), arr, lookup);
   k = PARENT(k);
 }
}
```

Outline

1. CertiGraph: Motivation and Overview

2. Mathematical and Spatial Representations

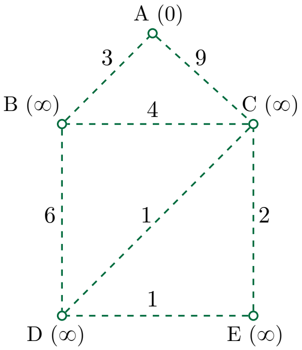
3. Shortest Path: Dijkstra

4. Minimum Spanning Forest: Prim and Kruskal

```
MST-PRIM(G,w,r):
 for each u in G.V
  /* elided: set up PQ,
      key, parent */
 r.key = 0
 while PQ ≠ ∅
  u = EXTRACT-MIN(PQ)
  for each v in G.Adj[u]
   if (v ∈ Q and
       w(u,v) < v.key)
    v.parent = u
    v.key = w(u,v)
```

Prim: missing the forest for the trees

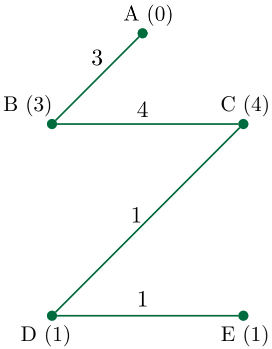
```
MST-PRIM(G,w,r):
 for each u in G.V
  /* elided: set up PQ,
     key, parent */
 r.key = 0
 while PQ ≠ ∅
  u = EXTRACT-MIN(PQ)
  for each v in G.Adj[u]
   if (v ∈ Q and
        w(u,v) < v.key)
    v.parent = u
    v.key = w(u,v)
```



Prim typically assumes a connected graph

Prim: missing the forest for the trees

```
MST-PRIM(G,w,r):
 for each u in G.V
  /* elided: set up PQ,
      key, parent */
 r.key = 0
 while PQ ≠ ∅
  u = EXTRACT-MIN(PQ)
  for each v in G.Adj[u]
   if (v ∈ Q and
        w(u,v) < v.key)
    v.parent = u
    v.key = w(u,v)
```



Prim typically assumes a connected graph

Prim: missing the forest for the trees

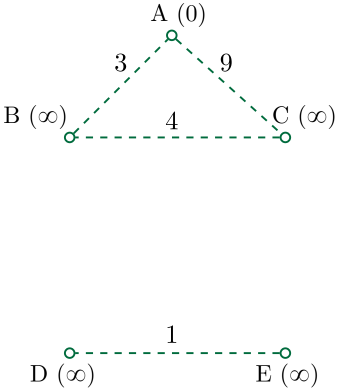
```
MST-PRIM(G,w,r):
 for each u in G.V
  /* elided: set up PQ,
     key, parent */
 r.key = 0
 while PQ ≠ ∅
  u = EXTRACT-MIN(PQ)
  for each v in G.Adj[u]
   if (v ∈ Q and
       w(u,v) < v.key)
    v.parent = u
    v.key = w(u,v)
```

How about an unconnected graph?

Prim: missing the forest for the trees

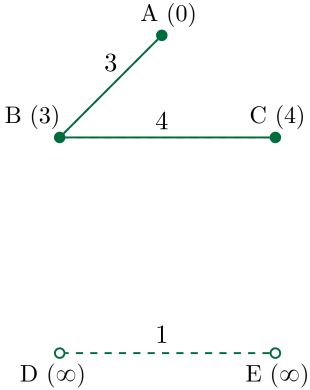
```
MST-PRIM(G,w,r):
 for each u in G.V
  /* elided: set up PQ,
     key, parent */
 r.key = 0
 while PQ ≠ ∅
  u = EXTRACT-MIN(PQ)
  for each v in G.Adj[u]
   if (v ∈ Q and
       w(u,v) < v.key)
    v.parent = u
    v.key = w(u,v)
```

How about an unconnected graph?

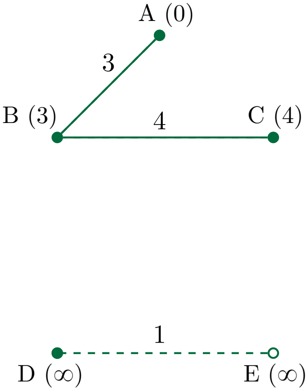


```
MST-PRIM(G,w,r):
 for each u in G.V
  /* elided: set up PQ,
     key, parent */
 r.key = 0
 while PQ ≠ ∅
  u = EXTRACT-MIN(PQ)
  for each v in G.Adj[u]
   if (v ∈ Q and
        w(u,v) < v.key)
    v.parent = u
    v.key = w(u,v)
```

How about an unconnected graph?



```
MST-PRIM(G,w,r):
 for each u in G.V
  /* elided: set up PQ,
     key, parent */
 r.key = 0
 while PQ ≠ ∅
  u = EXTRACT-MIN(PQ)
  for each v in G.Adj[u]
   if (v ∈ Q and
       w(u,v) < v.key)
    v.parent = u
    v.key = w(u,v)
```

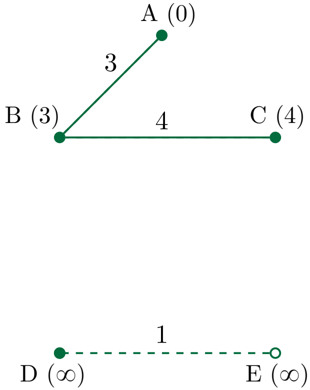


How about an unconnected graph?
D can now be extracted at cost ∞... (!)

Prim: missing the forest for the trees

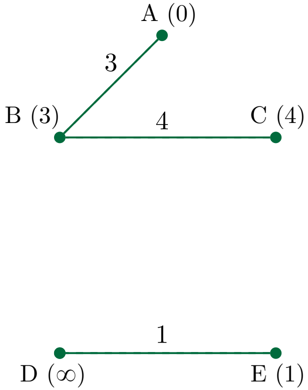
```
MST-PRIM(G,w,r):
 for each u in G.V
  /* elided: set up PQ,
     key, parent */
 r.key = 0
 while PQ ≠ ∅
  u = EXTRACT-MIN(PQ)
  for each v in G.Adj[u]
   if (v ∈ Q and
       w(u,v) < v.key)
    v.parent = u
    v.key = w(u,v)
```

How about an unconnected graph?
D can now be extracted at cost ∞... (!)
...meaning D is the root of a new tree!



Prim: missing the forest for the trees

```
MST-PRIM(G,w,r):
 for each u in G.V
  /* elided: set up PQ,
     key, parent */
 r.key = 0
 while PQ ≠ ∅
  u = EXTRACT-MIN(PQ)
  for each v in G.Adj[u]
   if (v ∈ Q and
        w(u,v) < v.key)
    v.parent = u
    v.key = w(u,v)
```



How about an unconnected graph?
D can now be extracted at cost ∞... (!)
...meaning D is the root of a new tree!

```
MST-PRIM(G,w,r):
 for each u in G.V
  u.key = INF
  u.parent = NIL
 r.key = 0
 Q = G.V
 while Q ≠ ∅
  u = EXTRACT-MIN(Q)
  for each v in G.Adj[u]
   if (v ∈ Q and
       w(u,v) < v.key)
    v.parent = u
    v.key = w(u,v)
```

Prim: an unnecessary argument; simpler invariants

```
MST-PRIM(G,w,r):
 for each u in G.V
  u.key = INF
  u.parent = NIL
 r.key = 0
 Q = G.V
 while Q ≠ ∅
  u = EXTRACT-MIN(Q)
  for each v in G.Adj[u]
   if (v ∈ Q and
       w(u,v) < v.key)
    v.parent = u
    v.key = w(u,v)
```

Prim: an unnecessary argument; simpler invariants

```
MST-PRIM(G,w,r):
 for each u in G.V
  u.key = INF
  u.parent = NIL
 r.key = 0
 Q = G.V
 while Q ≠ ∅
  u = EXTRACT-MIN(Q)
  for each v in G.Adj[u]
   if (v ∈ Q and
       w(u,v) < v.key)
    v.parent = u
    v.key = w(u,v)
```

```
MST-PRIM-NOROOT(G,w):
 for each u in G.V
  u.key = INF
  u.parent = NIL

 Q = G.V
 while Q ≠ ∅
  u = EXTRACT-MIN(Q)
  for each v in G.Adj[u]
   if (v ∈ Q and
       w(u,v) < v.key)
    v.parent = u
    v.key = w(u,v)
```

Contribution 8: machine-certified "real C" Prim

Prim: contributions

Contribution 8: machine-certified “real C” Prim

Contribution 9: more general specification

Contribution 8: machine-certified "real C" Prim

Contribution 9: more general specification

Contribution 6b: three adjacency matrix representations

Prim: contributions

Contribution 8: machine-certified “real C” Prim

Contribution 9: more general specification

Contribution 6b: three adjacency matrix representations
(code bases differ by less than 1%)

Prim: contributions

Contribution 8: machine-certified "real C" Prim

Contribution 9: more general specification

Contribution 6b: three adjacency matrix representations
(code bases differ by less than 1%)

Contribution 10: novel Prim variant without root

Kruskal: challenges and nonchallenges

Challenges:

Kruskal: challenges and nonchallenges

Challenges:

Edge list to represent the graph in memory

Kruskal: challenges and nonchallenges

Challenges:

- Edge list to represent the graph in memory
- Have to sort edge list

Kruskal: challenges and nonchallenges

Challenges:

- Edge list to represent the graph in memory
- Have to sort edge list
- Relies on union-find

Kruskal: challenges and nonchallenges

Challenges:

- Edge list to represent the graph in memory
- Have to sort edge list
- Relies on union-find
- Union-find itself uses a (directed) graph

Kruskal: challenges and nonchallenges

Challenges:

- Edge list to represent the graph in memory
- Have to sort edge list
- Relies on union-find
- Union-find itself uses a (directed) graph
- So must manipulate two graphs simultaneously

Kruskal: challenges and nonchallenges

Challenges:

- Edge list to represent the graph in memory
- Have to sort edge list
- Relies on union-find
- Union-find itself uses a (directed) graph
- So must manipulate two graphs simultaneously

Nonchallenges:

Challenges:

- Edge list to represent the graph in memory
- Have to sort edge list
- Relies on union-find
- Union-find itself uses a (directed) graph
- So must manipulate two graphs simultaneously

Nonchallenges:

- No algorithmic issues discovered...

Kruskal: challenges and nonchallenges

Challenges:

- Edge list to represent the graph in memory
- Have to sort edge list
- Relies on union-find
- Union-find itself uses a (directed) graph
- So must manipulate two graphs simultaneously

Nonchallenges:

- No algorithmic issues discovered...

Contribution 11: machine-certified “real C” Kruskal

Kruskal: challenges and nonchallenges

Challenges:
- Edge list to represent the graph in memory
- Have to sort edge list
- Relies on union-find
- Union-find itself uses a (directed) graph
- So must manipulate two graphs simultaneously

Nonchallenges:
- No algorithmic issues discovered...

Contribution 11: machine-certified "real C" Kruskal

Contribution 12: heapsort with $O(n)$ bottom-up `heapify`

Major contributions

Math, spatial support for adjacency matrices, edge lists

Major contributions

Math, spatial support for adjacency matrices, edge lists
Added support for undirected graphs

Major contributions

Math, spatial support for adjacency matrices, edge lists
Added support for undirected graphs

Dijkstra: nontrivial overflow

Major contributions

Math, spatial support for adjacency matrices, edge lists
Added support for undirected graphs

Dijkstra: nontrivial overflow
Prim: nontrivial improvement in spec

Major contributions

Math, spatial support for adjacency matrices, edge lists
Added support for undirected graphs

Dijkstra: nontrivial overflow
Prim: nontrivial improvement in spec
Kruskal: library handles directed + undirected

Major contributions

Math, spatial support for adjacency matrices, edge lists
Added support for undirected graphs

Dijkstra: nontrivial overflow
Prim: nontrivial improvement in spec
Kruskal: library handles directed + undirected
Binary heap: `decrease-key` and `heapify`, avoids overflow

Major contributions

Math, spatial support for adjacency matrices, edge lists
Added support for undirected graphs

Dijkstra: nontrivial overflow
Prim: nontrivial improvement in spec
Kruskal: library handles directed + undirected
Binary heap: `decrease-key` and `heapify`, avoids overflow

Development is modular and general
High effort—largely because we work with C—but good reuse

Major contributions

Math, spatial support for adjacency matrices, edge lists
Added support for undirected graphs

Dijkstra: nontrivial overflow
Prim: nontrivial improvement in spec
Kruskal: library handles directed + undirected
Binary heap: `decrease-key` and `heapify`, avoids overflow

Development is modular and general
High effort—largely because we work with C—but good reuse

Thanks!