Expanding CertiGraph: Dijkstra, Prim, and Kruskal

Anshuman Mohan, Wei Xiang Leow, Aquinas Hobor

National University of Singapore

NUS Programming Language and Verification Seminar
December 16, 2020

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

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

A Coq library to verify executable code
against realistic specifications
expressed with mathematical graphs

This Work

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

We verify Dijkstra, Prim, Kruskal

This Work

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

We verify Dijkstra, Prim, Kruskal

In doing so, we:

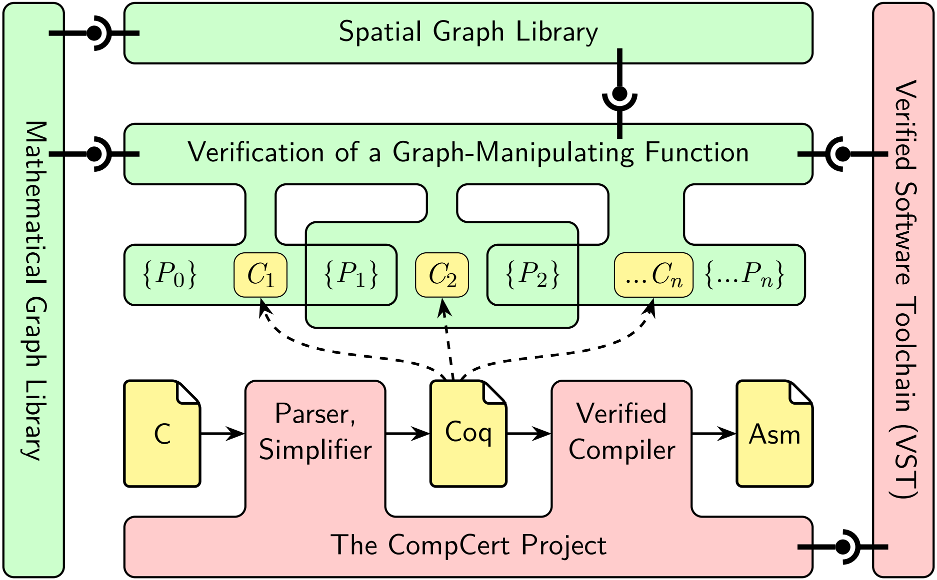
Test existing features [Dijk labels edges]
Expand into undirectedness [Prim, Krus]
Make nontrivial calls to verified methods [Krus calls UF]

Challenges

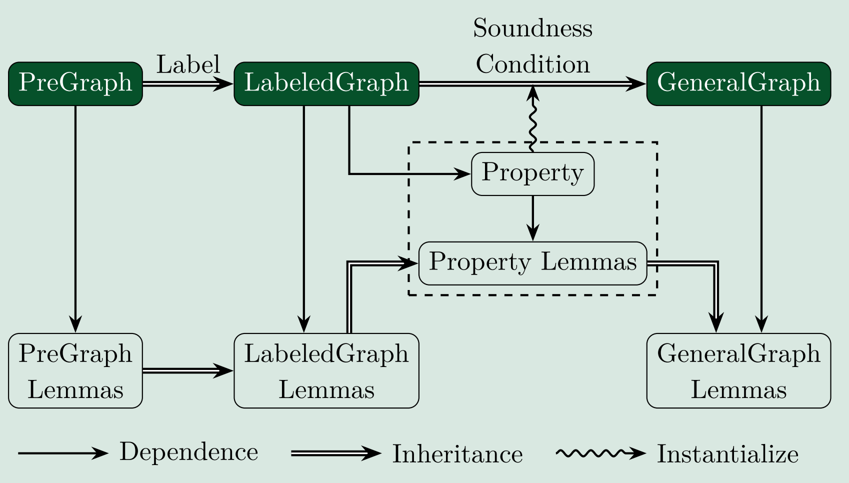
Using CompCert C, which
  is executable and realistic
  but has real-world complications

Aiming for full functional correctness

Maintaining modularity and reuse


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

Math Graph Architecture



A PreGraph is a hextuple ($\texttt{VType}, \texttt{EType}, \texttt{vvalid}, \texttt{evalid}, \texttt{src}, \texttt{dst}$)

$$
\textbf{Dijk\_PG}(\gamma) \stackrel{\text{def}}{=}
\begin{array}{l}
\texttt{VType := Z} \\
\texttt{EType := VType * VType} \\
\texttt{src := fst} \\
\texttt{dst := snd} \\
\forall v.\ \texttt{vvalid}(\gamma, v) \Leftrightarrow 0 \leqslant v < \texttt{size} \\
\forall s, d.\ \texttt{evalid}(\gamma, (s, d)) \Leftrightarrow \texttt{vvalid}(\gamma, s) \wedge \texttt{vvalid}(\gamma, d)
\end{array}
$$

Instantiating DijkGraph

A LabeledGraph is a quadruple (PreGraph, `VL`, `EL`, `GL`)

$$\mathbf{Dijk\_LG}(\gamma) \stackrel{\text{def}}{=} \text{Dijk\_PG as shown}$$

```
VL := list EL
EL := Z
GL := unit
```

Instantiating DijkGraph

A GeneralGraph adds arbitrary soundness conditions

$$
\begin{aligned}
\textbf{DijkGraph}(\gamma) \stackrel{\text{def}}{=}\ & \text{Dijk\_LG as shown, and} \\
& \mathit{FiniteGraph}(\gamma) \wedge \\
& \forall i, j.\ \texttt{vvalid}(\gamma, i) \wedge \texttt{vvalid}(\gamma, j) \Rightarrow \\
& \qquad i = j \Rightarrow \texttt{elabel}(\gamma, (i, j)) = 0 \wedge \\
& \qquad i \neq j \Rightarrow 0 \leqslant \texttt{elabel}(\gamma, (i, j)) \leqslant \lfloor \texttt{MAX/size} \rfloor \wedge \\
& \ldots
\end{aligned}
$$

A GeneralGraph adds arbitrary soundness conditions

$$\begin{aligned}\textbf{DijkGraph}(\gamma) \stackrel{\text{def}}{=} & \text{ Dijk\_LG as shown, and} \\ & \mathit{FiniteGraph}(\gamma) \wedge \\ & \forall i, j.\ \texttt{vvalid}(\gamma, i) \wedge \texttt{vvalid}(\gamma, j) \Rightarrow \\ & \qquad i = j \Rightarrow \texttt{elabel}(\gamma, (i, j)) = 0 \wedge \\ & \qquad i \neq j \Rightarrow 0 \leqslant \texttt{elabel}(\gamma, (i, j)) \leqslant \lfloor \texttt{MAX/size} \rfloor \wedge \\ & \ldots\end{aligned}$$

$$\mathsf{list\_rep}(\gamma, i) \stackrel{\text{def}}{=} \texttt{data\_at array graph2mat}(\gamma)[i]\ \texttt{list\_addr}(\gamma, i)$$

$$\mathsf{graph\_rep}(\gamma) \stackrel{\text{def}}{=} \mathop{\Huge\ast}_{\texttt{vvalid}(\gamma, v)} v \mapsto \mathsf{list\_rep}(\gamma, v)$$

Representing DijkGraph in Memory

$$\mathsf{list\_rep}(\gamma, i) \stackrel{\text{def}}{=} \texttt{data\_at array graph2mat}(\gamma)[i] \ \texttt{list\_addr}(\gamma, i)$$

$$\mathsf{graph\_rep}(\gamma) \stackrel{\text{def}}{=} \mathop{\text{\Large$\ast$}}_{\texttt{vvalid}(\gamma, v)} v \mapsto \mathsf{list\_rep}(\gamma, v)$$

Relies on restrictions placed at the Math level

```
void dijkstra (int **graph, int src, int *dist,
               int *prev, int size, int inf {
```

$\{\mathsf{AdjMat}(\gamma) * \mathsf{array}(\mathtt{dist}, \_) * \mathsf{array}(\mathtt{prev}, \_)\}$

Code and Specification

```
void dijkstra (int **graph, int src, int *dist,
               int *prev, int size, int inf {
```

$\{\mathsf{AdjMat}(\gamma) * \mathsf{array}(\mathtt{dist}, \_) * \mathsf{array}(\mathtt{prev}, \_)\}$

```
 int pq = init(size); int i, j, u, cost;
 for (i = 0; i < size; i++)
 {  dist[i] = inf; prev[i] = inf; push(i, inf, pq);  }
 dist[src] = 0; prev[src] = src; dec_key(src, 0, pq);
```

```
void dijkstra (int **graph, int src, int *dist,
               int *prev, int size, int inf {
```

$\{\mathsf{AdjMat}(\gamma) * \mathsf{array}(\texttt{dist}, \_) * \mathsf{array}(\texttt{prev}, \_)\}$

```
 int pq = init(size); int i, j, u, cost;
 for (i = 0; i < size; i++)
 {  dist[i] = inf; prev[i] = inf; push(i, inf, pq);  }
 dist[src] = 0; prev[src] = src; dec_key(src, 0, pq);
```

$$\left\{\begin{array}{l}\exists\, dist, prev, popped.\ \mathsf{AdjMat}(\gamma) * \mathsf{PQ}(\texttt{pq}) * \mathsf{array}(\texttt{dist}, dist) * \\ \quad \mathsf{array}(\texttt{prev}, prev) \wedge dijk\_correct(\gamma, \texttt{src}, popped, prev, dist)\end{array}\right\}$$

```
 // big while loop
```

Code and Specification

$$\left\{\begin{array}{l}\exists \mathit{dist}, \mathit{prev}, \mathit{popped}.\ \mathsf{AdjMat}(\gamma) * \mathsf{PQ}(\texttt{pq}) * \mathsf{array}(\texttt{dist}, \mathit{dist}) * \\ \quad \mathsf{array}(\texttt{prev}, \mathit{prev}) \wedge \mathit{dijk\_correct}(\gamma, \texttt{src}, \mathit{popped}, \mathit{prev}, \mathit{dist})\end{array}\right\}$$

Code and Specification

$$\left\{\begin{array}{l}\exists\, dist, prev, popped.\ \mathsf{AdjMat}(\gamma) * \mathsf{PQ}(\mathtt{pq}) * \mathsf{array}(\mathtt{dist}, dist) * \\ \ \mathsf{array}(\mathtt{prev}, prev) \wedge dijk\_correct(\gamma, \mathtt{src}, popped, prev, dist)\end{array}\right\}$$

```
while (!pq_emp(pq)) {
 u = popMin(pq);
```

Code and Specification

$$\left\{\begin{array}{l}\exists\, dist, prev, popped.\ \mathsf{AdjMat}(\gamma) * \mathsf{PQ}(\mathtt{pq}) * \mathsf{array}(\mathtt{dist}, dist) * \\ \ \mathsf{array}(\mathtt{prev}, prev) \wedge dijk\_correct(\gamma, \mathtt{src}, popped, prev, dist)\end{array}\right\}$$

```
while (!pq_emp(pq)) {
  u = popMin(pq);
```

$$// \left\{\begin{array}{l}\exists\, dist', prev', popped', i.\ \mathsf{AdjMat}(\gamma) * \mathsf{PQ}(\mathtt{pq}) * \\ \ \mathsf{array}(\mathtt{dist}, dist') * \mathsf{array}(\mathtt{prev}, prev') \wedge \\ \ dijk\_correct\_weak(\gamma, \mathtt{src}, popped', prev', dist', i, \mathtt{u})\end{array}\right\}$$

$$\left\{\begin{array}{l}\exists\, dist, prev, popped.\ \mathsf{AdjMat}(\gamma) * \mathsf{PQ}(\mathtt{pq}) * \mathsf{array}(\mathtt{dist}, dist) * \\ \ \mathsf{array}(\mathtt{prev}, prev) \wedge dijk\_correct(\gamma, \mathtt{src}, popped, prev, dist)\end{array}\right\}$$

```
while (!pq_emp(pq)) {
 u = popMin(pq);
```

$$// \left\{\begin{array}{l}\exists\, dist', prev', popped', i.\ \mathsf{AdjMat}(\gamma) * \mathsf{PQ}(\mathtt{pq}) * \\ \ \mathsf{array}(\mathtt{dist}, dist') * \mathsf{array}(\mathtt{prev}, prev') \wedge \\ \ dijk\_correct\_weak(\gamma, \mathtt{src}, popped', prev', dist', i, \mathtt{u})\end{array}\right\}$$

```
 for (i = 0; i < size; i++) {
 cost = getCell(graph, u, i);
 if (cost < inf) {
  if (dist[i] > dist[u] + cost) {
   dist[i] = dist[u] + cost; prev[i] = u;
   dec_key(i, dist[i], pq);
}}} // for
```

Code and Specification

$$\left\{\begin{array}{l}\exists \mathit{dist}, \mathit{prev}, \mathit{popped}.\ \mathsf{AdjMat}(\gamma) * \mathsf{PQ}(\mathtt{pq}) * \mathsf{array}(\mathtt{dist}, \mathit{dist}) * \\ \ \mathsf{array}(\mathtt{prev}, \mathit{prev}) \wedge \mathit{dijk\_correct}(\gamma, \mathtt{src}, \mathit{popped}, \mathit{prev}, \mathit{dist})\end{array}\right\}$$

```
while (!pq_emp(pq)) {
 u = popMin(pq);
```

$$// \left\{\begin{array}{l}\exists \mathit{dist'}, \mathit{prev'}, \mathit{popped'}, i.\ \mathsf{AdjMat}(\gamma) * \mathsf{PQ}(\mathtt{pq}) * \\ \ \mathsf{array}(\mathtt{dist}, \mathit{dist'}) * \mathsf{array}(\mathtt{prev}, \mathit{prev'}) \wedge \\ \ \mathit{dijk\_correct\_weak}(\gamma, \mathtt{src}, \mathit{popped'}, \mathit{prev'}, \mathit{dist'}, i, \mathtt{u})\end{array}\right\}$$

```
 for (i = 0; i < size; i++) {
 cost = getCell(graph, u, i);
 if (cost < inf) {
  if (dist[i] > dist[u] + cost) {
   dist[i] = dist[u] + cost; prev[i] = u;
   dec_key(i, dist[i], pq);
}}} // for
```

$$\left\{\begin{array}{l}\exists \mathit{dist''}, \mathit{prev''}.\ \mathsf{AdjMat}(\gamma) * \mathsf{PQ}(\mathtt{pq}) * \mathsf{array}(\mathtt{dist}, \mathit{dist''}) * \\ \ \mathsf{array}(\mathtt{prev}, \mathit{prev''}) \wedge \mathit{dijk\_correct}(\gamma, \mathtt{src}, \mathit{popped'}, \mathit{prev''}, \mathit{dist''})\end{array}\right\}$$

Code and Specification

$$\left\{\begin{array}{l}\exists dist, prev, popped.\ \mathsf{AdjMat}(\gamma) * \mathsf{PQ}(\mathtt{pq}) * \mathsf{array}(\mathtt{dist}, dist) * \\ \mathsf{array}(\mathtt{prev}, prev) \wedge dijk\_correct(\gamma, \mathtt{src}, popped, prev, dist)\end{array}\right\}$$

```
while (!pq_emp(pq)) {
 u = popMin(pq);
 for (i = 0; i < size; i++) {
 cost = getCell(graph, u, i);
 if (cost < inf) {
  if (dist[i] > dist[u] + cost) {
   dist[i] = dist[u] + cost; prev[i] = u;
   dec_key(i, dist[i], pq);
 }}} // for
```

Code and Specification

$$\left\{ \begin{array}{l} \exists \mathit{dist}, \mathit{prev}, \mathit{popped}.\ \mathsf{AdjMat}(\gamma) * \mathsf{PQ}(\mathtt{pq}) * \mathsf{array}(\mathtt{dist}, \mathit{dist}) * \\ \mathsf{array}(\mathtt{prev}, \mathit{prev}) \wedge \mathit{dijk\_correct}(\gamma, \mathtt{src}, \mathit{popped}, \mathit{prev}, \mathit{dist}) \end{array} \right\}$$

```
while (!pq_emp(pq)) {
 u = popMin(pq);
 for (i = 0; i < size; i++) {
 cost = getCell(graph, u, i);
 if (cost < inf) {
  if (dist[i] > dist[u] + cost) {
   dist[i] = dist[u] + cost; prev[i] = u;
   dec_key(i, dist[i], pq);
 }}} // for
} // while
```

$$\left\{\begin{array}{l}\exists \mathit{dist}, \mathit{prev}, \mathit{popped}.\ \mathsf{AdjMat}(\gamma) * \mathsf{PQ}(\mathtt{pq}) * \mathsf{array}(\mathtt{dist}, \mathit{dist}) * \\ \mathsf{array}(\mathtt{prev}, \mathit{prev}) \land \mathit{dijk\_correct}(\gamma, \mathtt{src}, \mathit{popped}, \mathit{prev}, \mathit{dist})\end{array}\right\}$$

```
while (!pq_emp(pq)) {
 u = popMin(pq);
 for (i = 0; i < size; i++) {
 cost = getCell(graph, u, i);
 if (cost < inf) {
  if (dist[i] > dist[u] + cost) {
   dist[i] = dist[u] + cost; prev[i] = u;
   dec_key(i, dist[i], pq);
 }}} // for
} // while
```

$$// \left\{\begin{array}{l}\exists \mathit{dist}^{\circ}, \mathit{prev}^{\circ}, \mathit{popped}^{\circ}.\ \mathsf{AdjMat}(\gamma) * \mathsf{PQ}(\mathtt{pq}) * \\ \mathsf{array}(\mathtt{dist}, \mathit{dist}^{\circ}) * \mathsf{array}(\mathtt{prev}, \mathit{prev}^{\circ}) \land \\ \mathit{all\_popped}(\mathit{popped}^{\circ}) \land \mathit{dijk\_correct}(\gamma, \mathtt{src}, \mathit{popped}^{\circ}, \mathit{prev}^{\circ}, \mathit{dist}^{\circ})\end{array}\right\}$$

Code and Specification

$$\left\{\begin{array}{l}\exists\, dist, prev, popped.\ \mathsf{AdjMat}(\gamma) * \mathsf{PQ}(\mathtt{pq}) * \mathsf{array}(\mathtt{dist}, dist) * \\ \ \mathsf{array}(\mathtt{prev}, prev) \wedge dijk\_correct(\gamma, \mathtt{src}, popped, prev, dist)\end{array}\right\}$$

```
while (!pq_emp(pq)) {
 u = popMin(pq);
 for (i = 0; i < size; i++) {
 cost = getCell(graph, u, i);
 if (cost < inf) {
  if (dist[i] > dist[u] + cost) {
   dist[i] = dist[u] + cost; prev[i] = u;
   dec_key(i, dist[i], pq);
 }}} // for
} // while
```

$$//\ \left\{\begin{array}{l}\exists\, dist^{\circ}, prev^{\circ}, popped^{\circ}.\ \mathsf{AdjMat}(\gamma) * \mathsf{PQ}(\mathtt{pq}) * \\ \ \mathsf{array}(\mathtt{dist}, dist^{\circ}) * \mathsf{array}(\mathtt{prev}, prev^{\circ}) \wedge \\ \ all\_popped(popped^{\circ}) \wedge dijk\_correct(\gamma, \mathtt{src}, popped^{\circ}, prev^{\circ}, dist^{\circ})\end{array}\right\}$$

```
freePQ (pq); return;
}
```

$$
\begin{aligned}
&\mathit{dijk\_correct}(\gamma, \mathit{src}, \mathit{popped}, \mathit{prev}, \mathit{dist}) \stackrel{\text{def}}{=} \\
&\quad \forall d.\ \mathit{vvalid}(\gamma, d) \Rightarrow \\
&\qquad \big(d \in \mathit{popped} \ \Rightarrow \\
&\qquad\qquad \exists \mathit{path}.\ \mathit{path\_correct}(\gamma, \mathit{prev}, \mathit{path}, \mathit{src}, d) \wedge \\
&\qquad\qquad \mathit{path\_globally\_optimal}(\gamma, \mathit{dist}, \mathit{path})\big) \wedge \\
&\quad \big(\mathit{dist}[d] < \texttt{inf} \ \Rightarrow \\
&\qquad\qquad \texttt{let}\ \ m\ \ \texttt{:=}\ \ \mathit{prev}[d]\ \ \texttt{in}\ \ m \in \mathit{popped}(\mathit{priq}) \wedge \\
&\qquad\qquad \forall m' \in \mathit{popped}(\mathit{priq}).\ \mathit{cost}(\mathit{p2m}\texttt{+}(m, d)) \leqslant \mathit{cost}(\mathit{p2m'}\texttt{+}(m', d))\big) \wedge \\
&\quad \big(\mathit{dist}[d] = \texttt{inf} \ \Rightarrow \\
&\qquad\qquad \forall m \in \mathit{popped}(\mathit{priq}).\ \mathit{cost}(\mathit{p2m}\texttt{+}(m, d)) = \texttt{inf}\big)
\end{aligned}
$$

dijk_correct

$$
\begin{aligned}
&\mathit{dijk\_correct}(\gamma, \mathit{src}, \mathit{popped}, \mathit{prev}, \mathit{dist}) \stackrel{\text{def}}{=} \\
&\quad \forall d.\ \mathit{vvalid}(\gamma, d) \Rightarrow \\
&\qquad \big(d \in \mathit{popped}\ \Rightarrow \\
&\qquad\qquad \exists \mathit{path}.\ \mathit{path\_correct}(\gamma, \mathit{prev}, \mathit{path}, \mathit{src}, d) \wedge \\
&\qquad\qquad \mathit{path\_globally\_optimal}(\gamma, \mathit{dist}, \mathit{path})\big) \wedge \\
&\quad \big(\mathit{dist}[d] < \texttt{inf}\ \Rightarrow \\
&\qquad\qquad \texttt{let}\ m\ \texttt{:=}\ \mathit{prev}[d]\ \texttt{in}\ m \in \mathit{popped}(\mathit{priq}) \wedge \\
&\qquad\qquad \forall m' \in \mathit{popped}(\mathit{priq}).\ \mathit{cost}(\mathit{p2m}\texttt{+}(m, d)) \leqslant \mathit{cost}(\mathit{p2m'}\texttt{+}(m', d))\big) \wedge \\
&\quad \big(\mathit{dist}[d] = \texttt{inf}\ \Rightarrow \\
&\qquad\qquad \forall m \in \mathit{popped}(\mathit{priq}).\ \mathit{cost}(\mathit{p2m}\texttt{+}(m, d)) = \texttt{inf}\big)
\end{aligned}
$$

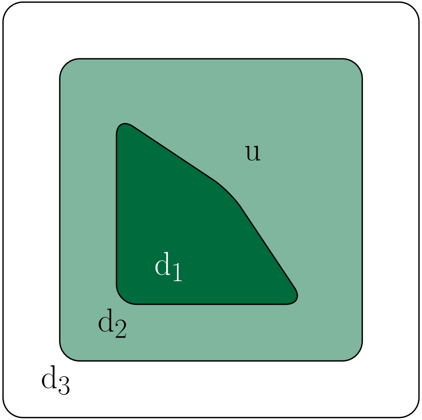
dijk_correct

$$
\begin{aligned}
&\mathit{dijk\_correct}(\gamma, \mathit{src}, \mathit{popped}, \mathit{prev}, \mathit{dist}) \stackrel{\text{def}}{=}\\
&\quad \forall d.\ \mathit{vvalid}(\gamma, d) \Rightarrow\\
&\qquad \big(d \in \mathit{popped} \ \Rightarrow\\
&\qquad\qquad \exists \mathit{path}.\ \mathit{path\_correct}(\gamma, \mathit{prev}, \mathit{path}, \mathit{src}, d) \wedge\\
&\qquad\qquad \mathit{path\_globally\_optimal}(\gamma, \mathit{dist}, \mathit{path})\big) \wedge\\
&\quad \big(\mathit{dist}[d] < \texttt{inf} \ \Rightarrow\\
&\qquad\qquad \texttt{let}\ m\ \texttt{:=}\ \mathit{prev}[d]\ \texttt{in}\ m \in \mathit{popped}(\mathit{priq}) \wedge\\
&\qquad\qquad \forall m' \in \mathit{popped}(\mathit{priq}).\ \mathit{cost}(\mathit{p2m}\texttt{+}(m, d)) \leqslant \mathit{cost}(\mathit{p2m'}\texttt{+}(m', d))\big) \wedge\\
&\quad \big(\mathit{dist}[d] = \texttt{inf} \ \Rightarrow\\
&\qquad\qquad \forall m \in \mathit{popped}(\mathit{priq}).\ \mathit{cost}(\mathit{p2m}\texttt{+}(m, d)) = \texttt{inf}\big)
\end{aligned}
$$

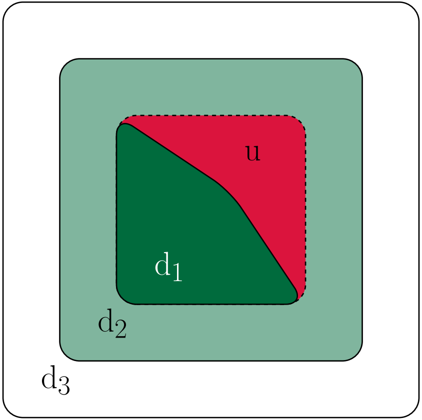
dijk_correct

$$
\begin{aligned}
& \mathit{dijk\_correct}(\gamma, \mathit{src}, \mathit{popped}, \mathit{prev}, \mathit{dist}) \stackrel{\text{def}}{=} \\
& \quad \forall d.\ \mathit{vvalid}(\gamma, d) \Rightarrow \\
& \qquad \big(d \in \mathit{popped} \ \Rightarrow \\
& \qquad\qquad \exists \mathit{path}.\ \mathit{path\_correct}(\gamma, \mathit{prev}, \mathit{path}, \mathit{src}, d) \land \\
& \qquad\qquad \mathit{path\_globally\_optimal}(\gamma, \mathit{dist}, \mathit{path})\big) \land \\
& \quad \big(\mathit{dist}[d] < \texttt{inf} \ \Rightarrow \\
& \qquad\qquad \texttt{let}\ m\ \texttt{:=}\ \mathit{prev}[d]\ \texttt{in}\ m \in \mathit{popped}(\mathit{priq}) \land \\
& \qquad\qquad \forall m' \in \mathit{popped}(\mathit{priq}).\ \mathit{cost}(\mathit{p2m}\texttt{+}(m, d)) \leqslant \mathit{cost}(\mathit{p2m'}\texttt{+}(m', d))\big) \land \\
& \quad \big(\mathit{dist}[d] = \texttt{inf} \ \Rightarrow \\
& \qquad\qquad \forall m \in \mathit{popped}(\mathit{priq}).\ \mathit{cost}(\mathit{p2m}\texttt{+}(m, d)) = \texttt{inf}\big)
\end{aligned}
$$

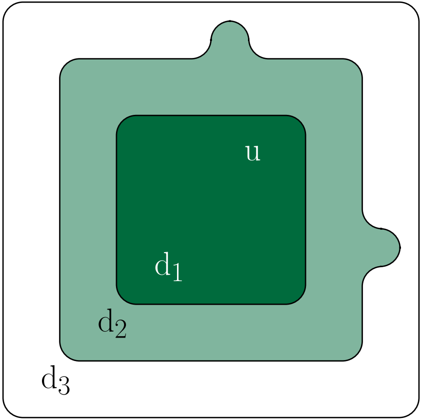
Key Transformation: Growing the Subgraph



Key Transformation: Growing the Subgraph



Key Transformation: Growing the Subgraph



dijk_correct_weak

$$
\begin{aligned}
& dijk\_correct\_weak(\gamma, src, popped, prev, dist, i, u) \stackrel{\text{def}}{=} \forall d. \\
& \quad \big(vvalid(\gamma, d) \;\Rightarrow\; d \in popped \;\Rightarrow\; \ldots \big) \wedge \\
& \quad \Big(0 \leqslant dst < i \;\Rightarrow\; \big(dist[d] < \texttt{inf} \;\Rightarrow \ldots \big) \wedge \big(dist[d] = \texttt{inf} \;\Rightarrow \ldots \big) \Big) \wedge \\
& \quad \Big(i \leqslant dst < size \;\Rightarrow \\
& \qquad \big(dist[d] < \texttt{inf} \;\Rightarrow \ldots \wedge m \neq u \wedge m' \neq u \big) \wedge \\
& \qquad \big(dist[d] = \texttt{inf} \;\Rightarrow \ldots \wedge m \neq u \big) \Big)
\end{aligned}
$$

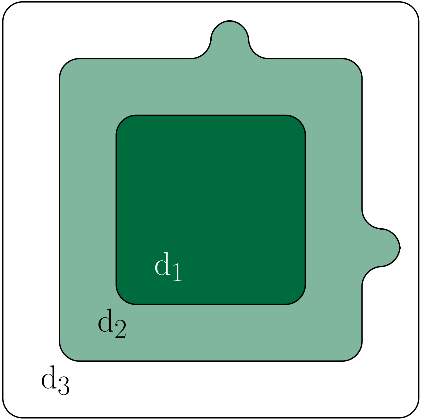
$$
\begin{aligned}
& \mathit{dijk\_correct\_weak}(\gamma, \mathit{src}, \mathit{popped}, \mathit{prev}, \mathit{dist}, i, u) \overset{\text{def}}{=} \forall d. \\
& \quad \left(\mathit{vvalid}(\gamma, d) \Rightarrow d \in \mathit{popped} \Rightarrow \ldots\right) \land \\
& \quad \Big(0 \leqslant \mathit{dst} < i \Rightarrow \left(\mathit{dist}[d] < \texttt{inf} \Rightarrow \ldots\right) \land \left(\mathit{dist}[d] = \texttt{inf} \Rightarrow \ldots\right)\Big) \land \\
& \quad \Big(i \leqslant \mathit{dst} < \mathit{size} \Rightarrow \\
& \qquad \left(\mathit{dist}[d] < \texttt{inf} \Rightarrow \ldots \land m \neq u \land m' \neq u\right) \land \\
& \qquad \left(\mathit{dist}[d] = \texttt{inf} \Rightarrow \ldots \land m \neq u\right)\Big)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{dijk\_correct\_weak}(\gamma, \mathit{src}, \mathit{popped}, \mathit{prev}, \mathit{dist}, i, u) \stackrel{\text{def}}{=} \forall d. \\
&\quad \big(\mathit{vvalid}(\gamma, d) \;\Rightarrow\; d \in \mathit{popped} \;\Rightarrow\; \ldots\big) \wedge \\
&\quad \Big(0 \leqslant \mathit{dst} < i \;\Rightarrow\; \big(\mathit{dist}[d] < \texttt{inf} \;\Rightarrow \ldots\big) \wedge \big(\mathit{dist}[d] = \texttt{inf} \;\Rightarrow \ldots\big)\Big) \wedge \\
&\quad \Big(i \leqslant \mathit{dst} < \mathit{size} \;\Rightarrow \\
&\qquad \big(\mathit{dist}[d] < \texttt{inf} \;\Rightarrow \ldots \wedge m \neq u \wedge m' \neq u\big) \wedge \\
&\qquad \big(\mathit{dist}[d] = \texttt{inf} \;\Rightarrow \ldots \wedge m \neq u\big)\Big)
\end{aligned}
$$

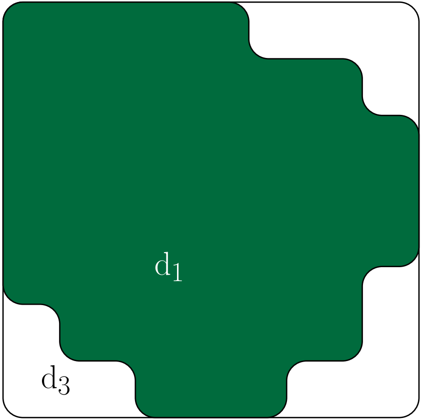
dijk_correct_weak

$$
\begin{aligned}
& \mathit{dijk\_correct\_weak}(\gamma, \mathit{src}, \mathit{popped}, \mathit{prev}, \mathit{dist}, i, u) \stackrel{\text{def}}{=} \forall d. \\
& \quad \big(\mathit{vvalid}(\gamma, d) \Rightarrow d \in \mathit{popped} \Rightarrow \ldots\big) \wedge \\
& \quad \Big(0 \leqslant \mathit{dst} < i \Rightarrow \big(\mathit{dist}[d] < \texttt{inf} \Rightarrow \ldots\big) \wedge \big(\mathit{dist}[d] = \texttt{inf} \Rightarrow \ldots\big)\Big) \wedge \\
& \quad \Big(i \leqslant \mathit{dst} < \mathit{size} \Rightarrow \\
& \qquad \big(\mathit{dist}[d] < \texttt{inf} \Rightarrow \ldots \wedge m \neq u \wedge m' \neq u\big) \wedge \\
& \qquad \big(\mathit{dist}[d] = \texttt{inf} \Rightarrow \ldots \wedge m \neq u\big)\Big)
\end{aligned}
$$

Postcondition



Postcondition



Overflow Strikes Again

The longest optimal path has `size-1` links

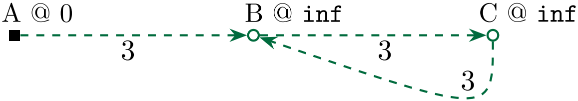
Overflow Strikes Again

The longest optimal path has `size-1` links
so say we set `elabel`'s upper bound to $\lfloor \texttt{MAX/(size-1)} \rfloor$

Overflow Strikes Again

The longest optimal path has `size-1` links
so say we set `elabel`'s upper bound to $\lfloor$`MAX/(size-1)`$\rfloor$

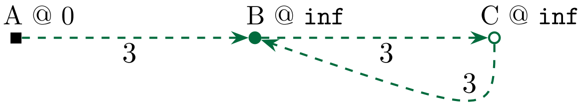
$\texttt{MAX} = 7$, $\texttt{size} = 3$, so $0 \leqslant \texttt{elabel}(\gamma, e) \leqslant 3$.



Overflow Strikes Again

The longest optimal path has `size-1` links
so say we set `elabel`'s upper bound to $\lfloor \texttt{MAX/(size-1)} \rfloor$

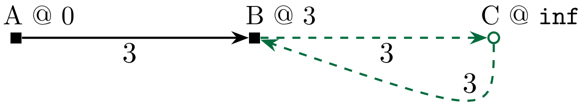
$\texttt{MAX} = 7$, $\texttt{size} = 3$, so $0 \leqslant \texttt{elabel}(\gamma, e) \leqslant 3$.



Overflow Strikes Again

The longest optimal path has `size-1` links
so say we set `elabel`'s upper bound to $\lfloor \texttt{MAX/(size-1)} \rfloor$

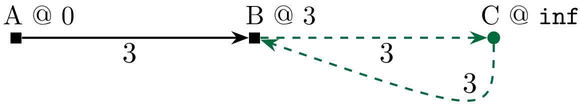
$\texttt{MAX} = 7$, $\texttt{size} = 3$, so $0 \leqslant \texttt{elabel}(\gamma, e) \leqslant 3$.



Overflow Strikes Again

The longest optimal path has `size-1` links
so say we set `elabel`'s upper bound to $\lfloor \texttt{MAX/(size-1)} \rfloor$

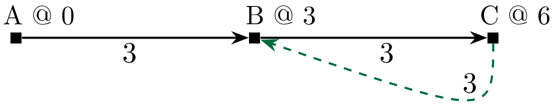
$\texttt{MAX} = 7$, $\texttt{size} = 3$, so $0 \leqslant \texttt{elabel}(\gamma, e) \leqslant 3$.



Overflow Strikes Again

The longest optimal path has `size-1` links
so say we set `elabel`'s upper bound to $\lfloor \texttt{MAX/(size-1)} \rfloor$

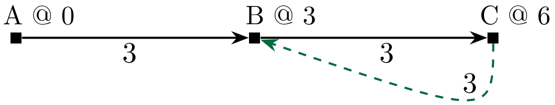
$\texttt{MAX} = 7$, $\texttt{size} = 3$, so $0 \leqslant \texttt{elabel}(\gamma, e) \leqslant 3$.



Overflow Strikes Again

The longest optimal path has `size-1` links
so say we set `elabel`'s upper bound to $\lfloor \texttt{MAX/(size-1)} \rfloor$

$\texttt{MAX} = 7$, $\texttt{size} = 3$, so $0 \leqslant \texttt{elabel}(\gamma, e) \leqslant 3$.

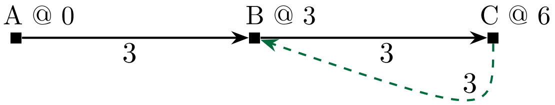


if $3 > 9$ then relax C $\rightsquigarrow$ B

Overflow Strikes Again

The longest optimal path has `size-1` links
so say we set `elabel`'s upper bound to $\lfloor$`MAX/(size-1)`$\rfloor$

`MAX` $= 7$, `size` $= 3$, so $0 \leqslant$ `elabel`$(\gamma, e) \leqslant 3$.

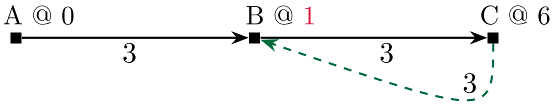


if $3 > 1$ then relax $\mathrm{C} \rightsquigarrow \mathrm{B}$

Overflow Strikes Again

The longest optimal path has `size-1` links
so say we set `elabel`'s upper bound to $\lfloor \texttt{MAX/(size-1)} \rfloor$

$\texttt{MAX} = 7$, $\texttt{size} = 3$, so $0 \leqslant \texttt{elabel}(\gamma, e) \leqslant 3$.

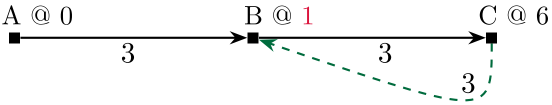


if $3 > 1$ then relax C $\rightsquigarrow$ B

Overflow Strikes Again

The longest optimal path has `size-1` links
so say we set `elabel`'s upper bound to $\lfloor \texttt{MAX/(size-1)} \rfloor$

$\texttt{MAX} = 7$, $\texttt{size} = 3$, so $0 \leqslant \texttt{elabel}(\gamma, e) \leqslant 3$.



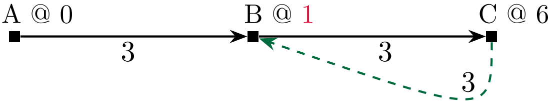
if $3 > 1$ then relax C $\rightsquigarrow$ B

One solution: Conservatively set upper bound to $\lfloor \texttt{MAX/size} \rfloor$

Overflow Strikes Again

The longest optimal path has `size-1` links
so say we set `elabel`'s upper bound to $\lfloor$`MAX/(size-1)`$\rfloor$

`MAX` $= 7$, `size` $= 3$, so $0 \leqslant$ `elabel`$(\gamma, e) \leqslant 3$.



if $3 > 1$ then relax C $\rightsquigarrow$ B

One solution: Conservatively set upper bound to $\lfloor$`MAX/size`$\rfloor$

Max path cost is then $\lfloor$`MAX/size`$\rfloor$ `* (size-1) = MAX -` $\lfloor$`MAX/size`$\rfloor$

Overflow Strikes Again

There are other ways to fix this!

Overflow Strikes Again

There are other ways to fix this!

Refactor troublesome addition as subtraction

Overflow Strikes Again

There are other ways to fix this!

- Refactor troublesome addition as subtraction
- Never look back into optimized part

Overflow Strikes Again

There are other ways to fix this!

- Refactor troublesome addition as subtraction
- Never look back into optimized part
- Your suggestion here

Overflow Strikes Again

There are other ways to fix this!

- Refactor troublesome addition as subtraction
- Never look back into optimized part
- Your suggestion here
- Your suggestion here

Overflow Strikes Again

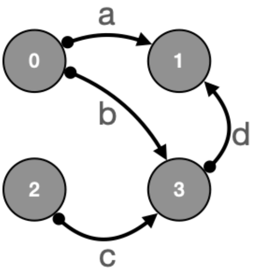
There are other ways to fix this!

- Refactor troublesome addition as subtraction
- Never look back into optimized part
- Your suggestion here
- Your suggestion here

Sadly, intuition supports `inf = MAX`

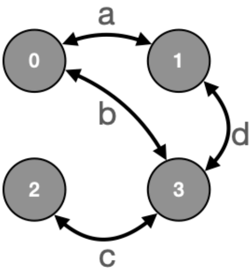
Prim: Extending to Undirectedness

Consider the AdjMat representation of a directed graph:



	0	1	2	3
0	0	a	inf	b
1	inf	0	inf	inf
2	inf	inf	0	c
3	inf	d	inf	0

Versus the AdjMat representation of an undirected graph:



	0	1	2	3
0	0	a	inf	b
1	a	0	inf	d
2	inf	inf	0	c
3	b	d	c	0

Prim: Extending to Undirectedness

Prevent double-counting:

```
Class SoundUAdjMat (g: UAdjMatLG) := {
  sadjmat: @SoundAdjMat size inf g;
  undirec: forall e, evalid g e -> src g e <= dst g e;
}.
```

Prevent double-counting:

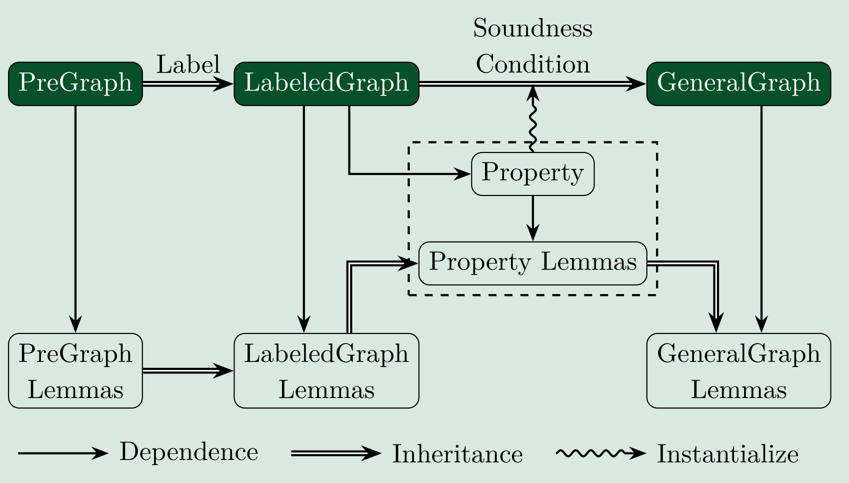
```
Class SoundUAdjMat (g: UAdjMatLG) := {
  sadjmat: @SoundAdjMat size inf g;
  undirec: forall e, evalid g e -> src g e <= dst g e;
}.
```

Build undirected idioms:

```
Definition adj_edge g e u v :=
  ((src g e = u /\ dst g e = v) \/
   (src g e = v /\ dst g e = u)).
```

Plus `upath`, `connected`, *etc.*

Recall: Math Graph Architecture



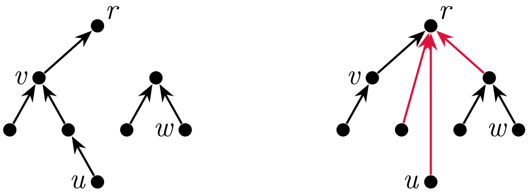
Kruskal: EdgeList Representation

Extend spatial support to accommodate EdgeList representation

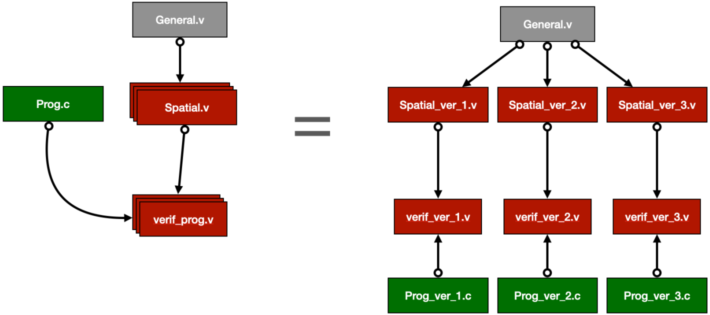
The double-counting restriction must be lifted:
an EdgeList-represented graph can have bona fide multi-connections

But the undirected idioms carry over

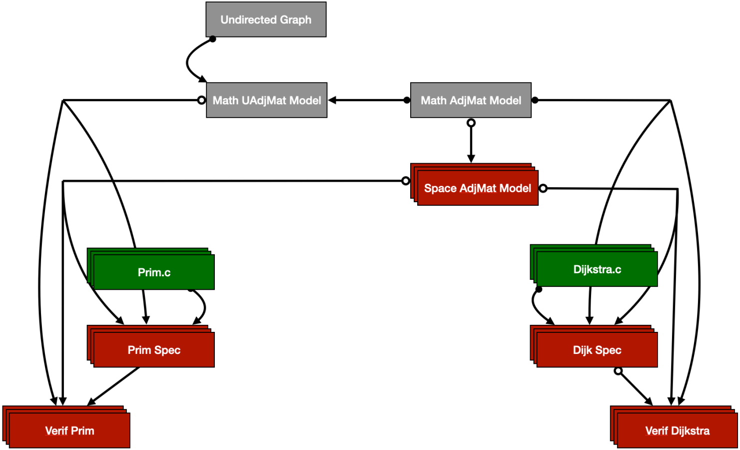
Consider performing `union u w`
Note: `reachable` is directed, `connected` is undirected



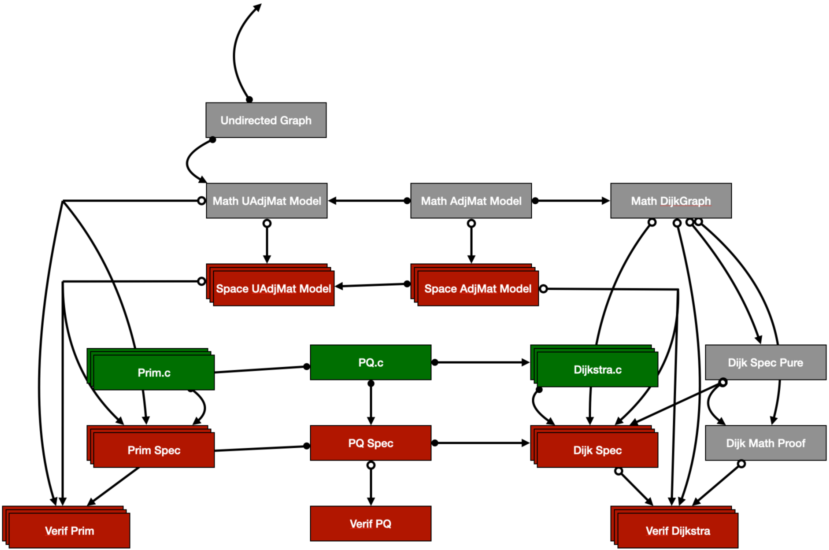
A Note on Modularity



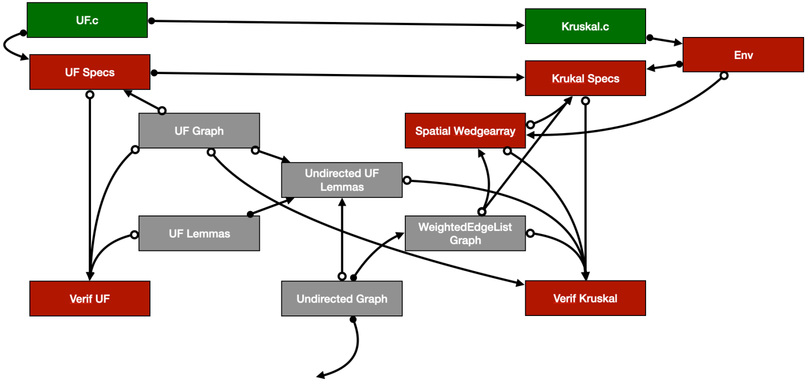
A Note on Modularity



A Note on Modularity



A Note on Modularity



Possible Next Steps

Verify PQ with `decrease-key`
AdjList representation for Dijkstra, Prim
Plug into verified `malloc`
Floyd-Warshall using AdjMat
Bellman-Ford using EdgeList

Possible Next Steps

Verify PQ with `decrease-key`
AdjList representation for Dijkstra, Prim
Plug into verified `malloc`
Floyd-Warshall using AdjMat
Bellman-Ford using EdgeList