本次作业的重点在于并行计算
你可以直接复用现有框架中的cost计算方法

cost-settings

Cost Settings in PathFinder Algorithms

1. Basic Pathfinder: Update cost
2. Optimized in VPR: Update cost
3. Basic Connection-based Routing
4. VPR8
5. CRoute
    5.1. HPWL version
    5.2. Timing-driven version
6. RWRoute
    6.1. Base cost $b(n)$
    6.2. Historical cost $h(n)$
    6.3. Present cost $p(n)$
    6.4. Sharing factor $1+share(n)$
    6.5. Node cost $c(n)$
    6.6. Upstream path cost $c_{prev}(n)$
    6.7. Estimated downstream cost $c_{exp}(n)$
7. Revisiting VPR8: dilute the influence of net order
8. Revisiting VPR8 again part 1:
9. Revisiting VPR8 again part 2 : consider critical net
10. BonnRoute
    10.1. Overview
    10.2. Formulation
    10.3. Algorithm
    10.4. Complexity
    10.5. Practical Implementation
        10.5.1. Early termination
        10.5.2. Solution reuse
        10.5.3. Parallelization

11. A* Algorithm

Cost Settings in PathFinder Algorithms

1. Basic Pathfinder: Update cost

> L. McMurchie and C. Ebeling, "Pathfinder: a negotiation-based performance-driven router for FPGAs," in FPGA 1995.

Node cost: $f(n) = b(n) \cdot h(n) \cdot p(n)$

$b(n)$: basic cost

$h(n)$: historical cost

$$h^{(k)}(n) = \begin{cases} 1 & k = 1 \\ h^{(k-1)}(n) & occ(n) \leq cap(n) \\ h^{(k-1)}(n) + h_f \cdot (occ(n) - cap(n)) & \text{otherwise} \end{cases}$$

$p(n)$: present congestion penalty

$$p(n) = \begin{cases} 1 & occ(n) < cap(n) \\ 1 + p_f \cdot (occ(n) - cap(n) + 1) & \text{otherwise} \end{cases}$$

2. Optimized in VPR: Update cost

> J. S. Swartz, V. Betz, and J. Rose, "A fast routability-driven router for FPGAs," in FPGA 1998.

Node cost: $f(n) = c_{prev} + b(n) \cdot h(n) \cdot p(n) + \alpha \cdot c_{exp}$

c_{prev} : cost of the previous wire nodes on the path from the source to this wire node

c_{exp} : expected cost of the path from this wire node to the sink node

$$c_{exp} = n_{ortho} \cdot b_{ortho} + n_{samedir} \cdot b_{samedir} + b_{ipin} + b_{sink}$$

3. Basic Connection-based Routing

> Vansteenkiste E, Bruneel K, Stroobandt D. "A connection-based router for FPGAs," in FPT 2013.

Node cost: $f(n) = c_{prev} + \dfrac{b(n) \cdot h(n) \cdot p(n)}{share(n)} + \alpha \cdot c_{exp}$

$share(n)$: #connections that legally share the node with the connection that SSSP algorithm currently searching for

$$c_{exp} = \frac{n_{ortho} \cdot b_{ortho}}{share(n)} + \frac{n_{samedir} \cdot b_{samedir}}{share(n)} + b_{ipin} + b_{sink}$$

Sometimes, $share(n)$ is replaced by $1 + share(n)$.

4. VPR8

> Murray K E, Petelin O, Zhong S, et al. "VTR 8: High-performance cad and customizable FPGA architecture modeling" in TRETS 2020.
>
> Murray K E, Zhong S, Betz V. "AIR: A fast but lazy timing-driven FPGA router." in ASP-DAC 2020.

Modifications: Line 25-26 in the Algorithm

- Net-based routing, connection-based ripup and reroute
- Ripup delay degraded nets (lazy)

5. CRoute

> Vercruyce D, Vansteenkiste E, Stroobandt D. "CRoute: A fast high-quality timing-driven connection-based FPGA router, " in FCCM 2019.

Modifications: Cost function based on Item 3

5.1. HPWL version

$$c_{exp}(n) = \frac{\delta_{same} \cdot \bar{c}_{same}}{1 + share(n)} + \frac{\delta_{ortho} \cdot \bar{c}_{ortho}}{1 + share(n)} + b_{ipin} + b_{sink}$$

$$c(n) = \frac{b(n) \cdot p(n) \cdot h(n)}{1 + share(n)} + c_{bias}(n), \quad c_{bias}(n) = \frac{b(n)}{2 \cdot fanout} \cdot \frac{\delta_{m,c}}{HPWL}$$

δ_{ortho} : the same as n_{ortho}

$\bar{c}_{ortho}/\bar{c}_{same}$: the average of a unit distance cost over all wire segments types

$\delta_{m,c}$: the manhattan distance to the geometric center of the net

5.2. Timing-driven version

$$c(n) = (1 - f_{crit}) \cdot c(n)_{wl} + f_{crit} \cdot T_{del}$$

$$f_{crit} = min\left[\left(1 - \frac{slack}{D_{max}}\right)^{\phi}, f_{crit,max}\right], \quad slack = T_{req} - T_{arr} - T_{del}$$

$$c_{exp}(n) = (1 - f_{crit}) \cdot \alpha \cdot c_{exp,wl}(n) + f_{crit} \cdot \beta \cdot c_{exp,td}(n)$$

$$c_{exp,td}(n) = \delta_{same} \cdot \bar{T}_{same} + \delta_{ortho} \cdot \bar{T}_{ortho}$$

$\bar{T}_{ortho}/\bar{T}_{same}$: average delay per distance in the same or ortho direction

6. RWRoute

> Zhou Y, Maidee P, Lavin C, et al. "RWRoute: An open-source timing-driven router for commercial FPGAs," in ACM TRETS 2021.

Modifications: Partial routing and parallel

$f(n) = c_{prev}(n) + c(n) + \alpha \cdot c_{exp}(n)$ with

- Upstream path cost $c_{prev}(n)$ in Section 6.6
- Estimated downstream cost $c_{exp}(n)$ in Section 6.7
- Node cost $c(n) = \dfrac{b(n) \cdot h(n) \cdot p(n)}{1 + share(n)} + c_{bias}(b(n))$ in Section 6.5
    - Base cost $b(n)$ in Section 6.1
    - Historical cost $h(n)$ in Section 6.2
    - Present cost $p(n)$ in Section 6.3
    - Sharing factor $1 + share(n)$ in Section 6.4
    - Bias cost $c_{bias}(b(n))$ also in Section 6.5

6.1. Base cost $b(n)$

$$b(n) = b_0(type(n)) \cdot L(n)$$

$type(n)$	$b_0(type(n))$	$L(n)$
`LAGUNA_I`	`0.0`	DontCare
`SUPER_LONG_LINE`	`0.3`	?
`WIRE` / `NODE_OUTPUT`	DontCare	`0`
`WIRE` / `NODE_CLE_OUTPUT`	DontCare	`0`
`WIRE` / `NODE_LAGUNA_OUTPUT`	DontCare	`0`

`WIRE` / `NODE_LAGUNA_DATA`	DontCare	`0`
`WIRE` / `NODE_LOCAL`	`0.4`	`1`
`WIRE` / `INTENT_DEFAULT`	`0.4`	`1`
`WIRE` / `NODE_SINGLE`	`0.4`	{ `1` , `2` }
`WIRE` / `NODE_DOUBLE`	`0.4`	（复杂）
`WIRE` / `NODE_HQUAD`	`0.35`	?
`WIRE` / `NODE_VQUAD`	`0.15`	{ `4` , `5` }
`WIRE` / `NODE_HLONG`	`0.15`	{ `6` , `7` }
`WIRE` / `NODE_VLONG`	`0.7`	`1`
`PINFEED_I`	`0.4`	`1`
`PINBOUND`	`0.4`	`1`
`PINFEED_O`	`1.0`	`1`

- default `0.4f`
- different node type with different value/computation, detailed in `setBaseCost()`

```java
// in "rwroute/RouteNode.java"
abstract public class RouteNode implements Comparable<RouteNode> {
    ...
    private void setBaseCost() {
        baseCost = 0.4f;
        switch (type) {
            case LAGUNA_I:
                // Make all approaches to SLLs zero-cost to encourage exploration
                // Assigning a base cost of zero would normally break congestion resolution
                // (since RWroute.getNodeCost() would return zero) but doing it here should be
                // okay because this node only leads to a SLL which will have a non-zero base cost
                baseCost = 0.0f;
                break;
            case SUPER_LONG_LINE:
                assert(length == RouteNodeGraph.SUPER_LONG_LINE_LENGTH_IN_TILES);
                baseCost = 0.3f * length;
                break;
            case WIRE:
                // NOTE: IntentCode is device-dependent
                IntentCode ic = node.getIntentCode();
                switch(ic) {
                    case NODE_OUTPUT:       // LUT route-thru
                    case NODE_CLE_OUTPUT:
                    case NODE_LAGUNA_OUTPUT:
                    case NODE_LAGUNA_DATA:  // US+: U-turn SLL at the boundary of the device
                        assert(length == 0);
                        break;
                    case NODE_LOCAL:
                    case INTENT_DEFAULT:
                        assert(length <= 1);
                        break;
                    case NODE_SINGLE:
                        assert(length <= 2);
                        if (length == 2) baseCost *= length;
                        break;
                    case NODE_DOUBLE:
                        if (endTileXCoordinate != node.getTile().getTileXCoordinate()) {
                            assert(length <= 2);

                            // Typically, length = 1 (since tile X is not equal)
                            // In US, have seen length = 2, e.g. VU440's INT_X171Y827/EE2_E_BEG7.
                            if (length == 2) baseCost *= length;
                        } else {
                            // Typically, length = 2 except for horizontal U-turns (length = 0)
                            // or vertical U-turns (length = 1).
                            // In US, have seen length = 3, e.g. VU440's INT_X171Y827/NN2_E_BEG7.
                            assert(length <= 3);
                        }
                        break;
                    case NODE_HQUAD:
                        assert (length != 0 || node.getAllDownhillNodes().isEmpty());
                        baseCost = 0.35f * length;
                        break;
                    case NODE_VQUAD:
                        // In case of U-turn nodes
                        if (length != 0) baseCost = 0.15f * length;// VQUADs have length 4 and 5
                        break;
                    case NODE_HLONG:
                        assert (length != 0 || node.getAllDownhillNodes().isEmpty());
                        baseCost = 0.15f * length;// HLONGs have length 6 and 7
                        break;
                    case NODE_VLONG:
                        baseCost = 0.7f;
                        break;
                    default:
                        throw new RuntimeException(ic.toString());
                }
                break;
            case PINFEED_I:
            case PINBOUNCE:
                break;
            case PINFEED_O:
                baseCost = 1f;
                break;
            default:
                throw new RuntimeException(type.toString());
        }
    }
```

6.2. Historical cost $h(n)$

$$h(n) \leftarrow h(n) + h_f \cdot \max\{0, occ(n) - cap(n)\}$$

- $occ(n)$: occupancy of route node n
- $cap(n)$: capacity of route node n

```java
// in "rwroute/RouteNode.java"
abstract public class RouteNode implements Comparable<RouteNode> {
    ...
    public static final int initialHistoricalCongestionCost = 1;
    ...
}

// in "rwroute/RwRoute.java"
public class RWRoute {
    ...
    private void updateCost() {
        overUsedRnodes.clear();
        for (RouteNode rnode : routingGraph.getRnodes()) {
            int overuse=rnode.getOccupancy() - RouteNode.capacity;
            if (overuse == 0) {
                rnode.setPresentCongestionCost(1 + presentCongestionFactor);
            } else if (overuse > 0) {
                overUsedRnodes.add(rnode);
                rnode.setPresentCongestionCost(1 + (overuse + 1) * presentCongestionFactor);
                rnode.setHistoricalCongestionCost(rnode.getHistoricalCongestionCost() + overuse * historicalCongestionFactor);
            } else {
                assert(overuse < 0);
                assert(rnode.getPresentCongestionCost() == 1);
            }
        }
    }
    ...
}
```

- historicalCongestionFactor h_f
    - historicalCongestionFactor: `1f`
    - can be set by users

```java
// in "rwroute/RWRouteConfig.java"
public class RWRouteConfig {
    ...
    public RWRouteConfig(String[] arguments) {
        ...
        historicalCongestionFactor = 1f;
        ...
    }
    ...
}

// in "rwroute/RwRoute.java"
public class RWRoute {
    ...
    private void initializeRouting() {
        ...
        historicalCongestionFactor = config.getHistoricalCongestionFactor();
        ...
    }
}
```

6.3. Present cost $p(n)$

$$p(n) = \begin{cases} 1 & occ(n) < cap(n) \\ 1 + p_f \cdot (occ(n) - cap(n) + 1) & occ(n) \geq cap(n) \end{cases}$$

```java
// in "rwroute/RouteNode.java"
abstract public class RouteNode implements Comparable<RouteNode> {
    ...
    /**
     * Updates the present congestion cost based on the present congestion penalty factor.
     * @param pres_fac The present congestion penalty factor.
     */
    public void updatePresentCongestionCost(float pres_fac) {
        int occ = getOccupancy();
        int cap = RouteNode.capacity;

        if (occ < cap) {
            setPresentCongestionCost(1);
        } else {
            setPresentCongestionCost(1 + (occ - cap + 1) * pres_fac);
        }
    }
    ...
}

// in "rwroute/RwRoute.java"
public class RWRoute {
    ...
    private void updateCost() {
        overUsedRnodes.clear();
        for (RouteNode rnode : routingGraph.getRnodes()) {
            int overuse=rnode.getOccupancy() - RouteNode.capacity;
            if (overuse == 0) {
                rnode.setPresentCongestionCost(1 + presentCongestionFactor);
            } else if (overuse > 0) {
                overUsedRnodes.add(rnode);
                rnode.setPresentCongestionCost(1 + (overuse + 1) * presentCongestionFactor);
                rnode.setHistoricalCongestionCost(rnode.getHistoricalCongestionCost() + overuse * historicalCongestionFactor);
            } else {
                assert(overuse < 0);
                assert(rnode.getPresentCongestionCost() == 1);
            }
        }
    }
    ...
}
```

- presentCongestionFactor p_f
    - $p_f^{(k)} = 0.5 \times 2^k$ for $k = 0, 1, 2, \cdots$
    - initialPresentCongestionFactor: `0.5f`
    - presentCongestionMultiplier: `2f`

```java
// in "rwroute/RWRouteConfig.java"
public class RWRouteConfig {
    ...
    public RWRouteConfig(String[] arguments) {
        ...
        initialPresentCongestionFactor = 0.5f;
        presentCongestionMultiplier = 2f;
        ...
    }
    ...
}

// in "rwroute/RwRoute.java"
public class RWRoute {
    ...
    private void initializeRouting() {
        ...
        presentCongestionFactor = config.getInitialPresentCongestionFactor
();
        ...
    }
    ...
    protected void initialize() {
        ...
        presentCongestionFactor = config.getInitialPresentCongestionFactor
();
        ...
    }
    ...
    private void updateCostFactors() {
        updateCongestionCosts.start();
        presentCongestionFactor *= config.getPresentCongestionMultiplier
();
        updateCost();
        updateCongestionCosts.stop();
    }
    ...
}
```

6.4. Sharing factor $1 + share(n)$

- $share(n) = shareWeight(n) \cdot \texttt{countSourceUses}$
- $shareWeight(n) = (1 - f_{crit}(c))^2$
  - 2 : `shareExponent`
  - $\lambda(c) = 1 - f_{crit}(c)$: `rnodeCostWeight`
    - The connection-dependent value $\lambda(c)$ will also be used in Section 6.6 and 6.7.
- $f_{crit}^{(k)}(c) = \begin{cases} 0 & (k = 0) \text{ or } (\texttt{timingDriven == false}) \\ \max\{f_{crit}^{(k-1)}(c), f_{crit,temp}(c)\} & (k \geq 1) \text{ and } (\texttt{timingDriven == true}) \end{cases}$
- $f_{crit,temp}(c) = \left(1 - \frac{slack}{D_{max}}\right)^{\phi} \cdot f_{crit,max}$
  - $\phi = 3$: `criticalityExponent`
  - $f_{crit,max} = 0.99$: `MAX_CRITICALITY`
  - $slack = \min\limits_{(u,v) \in E} \{T_{req}(v) - T_{arr}(u) - T_{delay}(u, v)\}$
- 论文里： $f_{crit} = min\left[\left(1 - \frac{slack}{D_{max}}\right)^{\phi}, f_{crit,max}\right]$

```java
public class RWRoute {
    ...
    private static final float MAX_CRITICALITY = 0.99f;
    ...
    private void preRoutingEstimation() {
        if (config.isTimingDriven()) {
            ...
            timingManager.calculateCriticality(indirectConnections, MAX_CRITICALITY, config.getCriticalityExponent());
            ...
        }
    }
    ...
    protected void routeConnection(Connection connection) {
        float rnodeCostWeight = 1 - connection.getCriticality();
        float shareWeight = (float) (Math.pow(rnodeCostWeight, config.getShareExponent()));
        ...
    }
    ...
    protected void evaluateCostAndPush(...) {
        ...
        float sharingFactor = 1 + sharingWeight * countSourceUses;
        ...
    }
    ...
}

public class RWRouteConfig {
    ...
    public RWRouteConfig(String[] arguments) {
        ...
        shareExponent = 2;
        criticalityExponent = 3;
        ...
    }
    ...
}

public class Connection implements Comparable<Connection> {
    ...
    public Connection(...) {
        ...
        criticality = 0f;
        ...
    }
    ...
    public void calculateCriticality(float maxDelay, float maxCriticality, float criticalityExponent) {
        float minSlack = Float.MAX_VALUE;
        for (TimingEdge e : getTimingEdges()) {
            float slack = e.getDst().getRequiredTime() - e.getSrc().getArrivalTime() - e.getDelay();
            minSlack = Float.min(minSlack, slack);
        }

        // Negative slacks are not supported, and should not occur if maxDelay was
        // normalized correctly.
        assert(minSlack >= 0);

        float tempCriticality  = (1 - minSlack / maxDelay);

        tempCriticality = (float) Math.pow(tempCriticality, criticalityExponent) * maxCriticality;

        if (tempCriticality > criticality)
            setCriticality(tempCriticality);
    }
    ...
}

public class TimingManager {
    ...
    public void calculateCriticality(List<Connection> connections, float maxCriticality, float criticalityExponent) {
        for (Connection connection:connections) {
            connection.resetCriticality();
        }
        float maxRequired = timingGraph.superSink.getRequiredTime();
        for (Connection connection : connections) {
            connection.calculateCriticality(maxRequired, maxCriticality, criticalityExponent);
        }
    }
    ...
}
```

6.5. Node cost $c(n)$

$$c(n) = \frac{b(n) \cdot h(n) \cdot p(n)}{1 + share(n)} + c_{bias}(b(n))$$

- $c_{bias}(b(n)) = \frac{b(n)}{fanout} \cdot \frac{\delta_{m,c}}{2 \cdot HPWL}$
- $\delta_{m,c} = |\texttt{rnode.EndTileX} - \texttt{net.XCenter}| + |\texttt{rnode.EndTileY} - \texttt{net.YCenter}|$

```java
public class RWRoute {
    ...
    private float getNodeCost(RouteNode rnode, Connection connection, int countSameSourceUsers, float sharingFactor) {
        boolean hasSameSourceUsers = countSameSourceUsers!= 0;
        float presentCongestionCost;

        if (hasSameSourceUsers) {// the rnode is used by other connection(s) from the same net
            int overoccupancy = rnode.getOccupancy() - RouteNode.capacity;
            // make the congestion cost less for the current connection
            presentCongestionCost = 1 + overoccupancy * presentCongestionFactor;
        } else {
            presentCongestionCost = rnode.getPresentCongestionCost();
        }

        float biasCost = 0;
        if (!rnode.isTarget() && rnode.getType() != RouteNodeType.SUPER_LONG_LINE) {
            NetWrapper net = connection.getNetWrapper();
            biasCost = rnode.getBaseCost() / net.getConnections().size() *
                    (Math.abs(rnode.getEndTileXCoordinate() - net.getXCenter()) + Math.abs(rnode.getEndTileYCoordinate() - net.getYCenter())) / net.getDoubleHpwl();
        }

        return rnode.getBaseCost() * rnode.getHistoricalCongestionCost() * presentCongestionCost / sharingFactor + biasCost;
    }
    ...
}
```

6.6. Upstream path cost $c_{prev}(n)$

Let node n_{prev} be node n's previous node on the path $s \rightsquigarrow n$:

$$c_{prev}(n) = c_{prev}(n_{prev}) + \lambda(c) \cdot c(n) + \lambda(c) \cdot (1-\alpha) \cdot \frac{L(n)}{1 + share(n_{prev})}$$

- $\lambda(c) = 1 - f_{crit}(c)$: `rnodeCostWeight`, see Section 6.4

- α : `wlWeight` = `0.8f`
- $\lambda(c) \cdot (1 - \alpha)$: `rnodeLengthWeight` = `rnodeCostWeight` * (1 – `wlWeight`)

```java
public class RWRoute {
    ...
    protected void evaluateCostAndPush(...) {⋯}
    ...
    protected void push(RouteNode childRnode, float newPartialPathCost, float newTotalPathCost) {
        ...
        childRnode.setUpstreamPathCost(newPartialPathCost);
        ...
    }
    ...
}

public void setUpstreamPathCost(float newPartialPathCost) {
    this.upstreamPathCost = newPartialPathCost;
}
```

6.7. Estimated downstream cost $c_{exp}(n)$

$$c_{exp}(n) = \lambda(c) \cdot \alpha \cdot \frac{\delta_x + \delta_y}{1 + share(n)}$$

- $\delta_x + \delta_y$: `deltaX + deltaY`
    - `deltaX` = Math.abs(childX – sinkX)
    - `deltaY` = Math.abs(childY – sinkY)
- $\lambda(c) \cdot \alpha$: `rnodeEstWlWeight`
    - `rnodeEstWlWeight` = `rnodeCostWeight` * `wlWeight`
    - $\lambda(c)$: `rnodeCostWeight` , see Section 6.4
    - α : `wlweight` = `0.8f`
- b_{ipin} , b_{sink} ：论文里说，不重要，没加到cost里

> The b_{ipin} and b_{sink} are not expressed explicitly, because the input pin and the sink are not part of the routing expansion.

```java
public class RWRoute {
    ...
    protected void evaluateCostAndPush(RouteNode rnode, boolean longParent, RouteNode childRnode, Connection connection, float sharingWeight, float rnodeCostWeight,
                                       float rnodeLengthWeight, float rnodeEstWlWeight,
                                       float rnodeDelayWeight, float rnodeEstDlyWeight) {
        int countSourceUses = childRnode.countConnectionsOfUser(connection.getNetWrapper());
        float sharingFactor = 1 + sharingWeight* countSourceUses;

        // Set the prev pointer, as RouteNode.getEndTileYCoordinate() and
        // RouteNode.getSLRIndex() require this
        childRnode.setPrev(rnode);

        float newPartialPathCost = rnode.getUpstreamPathCost() + rnodeCostWeight * getNodeCost(childRnode, connection, countSourceUses, sharingFactor)
                                + rnodeLengthWeight * childRnode.getLength() / sharingFactor;
        if (config.isTimingDriven()) {
            newPartialPathCost += rnodeDelayWeight * (childRnode.getDelay() + DelayEstimatorBase.getExtraDelay(childRnode.getNode(), longParent));
        }

        int childX = childRnode.getEndTileXCoordinate();
        int childY = childRnode.getEndTileYCoordinate();
        RouteNode sinkRnode = connection.getSinkRnode();
        int sinkX = sinkRnode.getBeginTileXCoordinate();
        int sinkY = sinkRnode.getBeginTileYCoordinate();
        int deltaX = Math.abs(childX - sinkX);
        int deltaY = Math.abs(childY - sinkY);
        ...

        int distanceToSink = deltaX + deltaY;
        float newTotalPathCost = newPartialPathCost + rnodeEstWlWeight * distanceToSink / sharingFactor;
        if (config.isTimingDriven()) {
            newTotalPathCost += rnodeEstDlyWeight * (deltaX * 0.32 + deltaY * 0.16);
        }
        push(childRnode, newPartialPathCost, newTotalPathCost);
    }
    ...
}
```

child

```java
public class RouteNodeInfo {
    ...
    public static RouteNodeInfo get(Node node) {
        Wire[] wires = node.getAllWiresInNode();
        Tile baseTile = node.getTile();
        TileTypeEnum baseType = baseTile.getTileTypeEnum();
        Tile endTile = null;
        boolean pinfeedIntoLaguna = false;
        for (Wire w : wires) {
            Tile tile = w.getTile();
            TileTypeEnum tileType = tile.getTileTypeEnum();
            boolean lagunaTile = false;
            if (tileType == TileTypeEnum.INT ||
                    (lagunaTile = Utils.isLaguna(tileType))) {
                if (!lagunaTile ||
                        // Only consider a Laguna tile as an end tile if base tile is Laguna too
                        // (otherwise it's a PINFEED into a Laguna)
                        Utils.isLaguna(baseType)) {
                    boolean endTileWasNotNull = (endTile != null);
                    endTile = tile;
                    // Break if this is the second INT tile
                    if (endTileWasNotNull) break;
                } else {
                    assert(!Utils.isLaguna(baseType));
                    pinfeedIntoLaguna = (node.getIntentCode() == IntentCode.NODE_PINFEED);
                }
            }
        }
        if (endTile == null) {
            endTile = node.getTile();
        }

        RouteNodeType type = getType(node, endTile, pinfeedIntoLaguna);
        short endTileXCoordinate = getEndTileXCoordinate(node, type, (short) endTile.getTileXCoordinate());
        short endTileYCoordinate = (short) endTile.getTileYCoordinate();
        short length = getLength(baseTile, type, endTileXCoordinate, endTileYCoordinate);

        return new RouteNodeInfo(type, endTileXCoordinate, endTileYCoordinate, length);
    }
    ...
}
```

7. Revisiting VPR8: dilute the influence of net order

> Zha Y, Li J. "Revisiting PathFinder routing algorithm," in FPGA 2022.

Modifications: p_f as constant value

8. Revisiting VPR8 again part 1:

$S_i = \{Path_i^j\}$: All possible routing paths for net i sorted by the cost of path in the ascending order

$Path_i^0$: routed path of net i

congestion: $Path_i^0 \cap Path_j^0 \neq \emptyset$

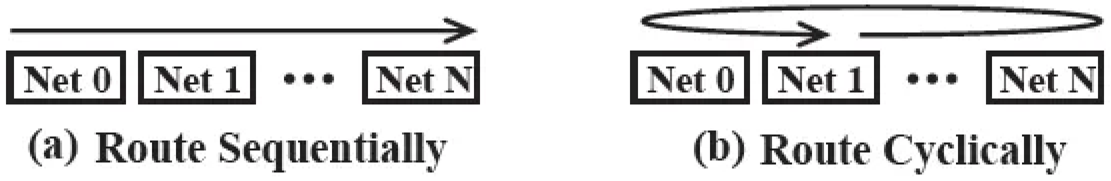
routable: $\exists\, k_i, k_j\ such\ that\ Path_i^{k_i} \cap Path_j^{k_j} = \emptyset$

If $S_i \neq S_j$:

- find smallest $k_i\ and\ k_j$: $Path_i^{k_i} \cap Path_j^{k_j} = \emptyset$
- increase cost of routing nodes so that $\forall k \neq k_i,\ Cost(Path_i^k) > Cost(Path_i^{k_i})$ or $\forall k \neq k_j,\ Cost(Path_j^k) > Cost(Path_j^k)$

If $S_i == S_j$: i和j的权重更新将会同步，所以，永远只和net order相关

- 循环（cyclical routing）布线: 一个周期，net循环布，减轻order的影响



- 修改 p_{base} ： 下面一篇文章

runtime评估：1.37x/2.39x speed up on average compared with paper 7 and VPR 8.

9. Revisiting VPR8 again part 2 : consider critical net

$cost_{path} = A_{ij} \times delay_{path} + (1 - A_{ij}) \times cong_{path}$

- A_{ij} : $0 \leq A_{ij} \leq 1$, slack ratio, 越大表示这条net越重要，

- $cong_{path} = \sum_n (d_n + h_n) \times p_n$
    - d_n : routing delay of the node
    - h_n ： congestion of the node

$$astar\_fac = 1.2 + min(\frac{N_{iter}}{50}, 2) \times max(0, 1 - \frac{A_{ij}}{0.8})$$

$$p_{base}^{iter} = \begin{cases} min(N_{iter} \times 0.05, 15) & A_{ij} \geq 0.8 \\ max(1, min(N_{iter} \times 0.05, 15)) & else \end{cases}$$

runtime评估：2.63x/4.15x speed up on average compared with paper 7 and VPR 8.

10. BonnRoute

10.1. Overview

> [MRV11] D. Müller, K. Radke, and J. Vygen, "Faster min—max resource sharing in theory and practice," Math. Program. Comput., vol. 3, no. 1, pp. 1—35, Mar. 2011, doi: 10.1007/s12532-011-0023-y.
>
> [GMN13] M. Gester, D. Müller, T. Nieberg, C. Panten, C. Schulte, and J. Vygen, "BonnRoute: Algorithms and Data Structures for Fast and Good VLSI Routing," ACM Trans. Des. Autom. Electron. Syst., vol. 18, no. 2, pp. 1—24, Mar. 2013, doi: 10.1145/2442087.2442103.
>
> [Bla22] D. Blankenburg, "Resource Sharing Revisited: Local Weak Duality and Optimal Convergence," in 30th Annual European Symposium on Algorithms (ESA 2022), 2022, vol. 244, pp. 20:1-–20:14, doi: 10.4230/LIPIcs.ESA.2022.20.
>
> [Sac22] P. Saccardi, "Continuous Routing," PhD dissertation, University Bonn, 2022.

The main idea of [MRV11][GMN13][Bla22][Sac22] is to cast a routing problem (VLSI global routing) into a *min-max resource sharing problem (MMRSP)*, where multiple consumers compete for multiple resources. Given an approximate oracle (a solver for a single consumer), an algorithm of MMRSP exists that calls this orale within polynomial times and generates approximate solutions.

10.2. Formulation

> **Min-Max Resource Sharing Problem (MMRSP)**: You are given a resource set $\mathcal{V}$ and a consumer set $\mathcal{N}$; and for every $n \in \mathcal{N}$, you are also given a convex block of feasible solutions

> $\mathcal{B}_n$ and a convex usage function $g_n : \mathcal{B}_n \to \mathbb{R}_+^{\mathcal{V}}$. Find $b_n \in \mathcal{B}_n$ for every $n \in \mathcal{N}$ such that the largest resource consumption is minimized, i.e., find
>
> $$\mathbf{b}^* = (b_n^*) = \underset{b_n \in \mathcal{B}_n \text{ for } n \in \mathcal{N}}{\arg\min} \left\{ \max_{v \in \mathcal{V}} \sum_{n \in \mathcal{N}} (g_n(b_n))_v \right\}$$
>
> and $\lambda^* = \max_{v \in \mathcal{V}} \sum_{n \in \mathcal{N}} (g_n(b_n^*))_v$.

In FPGA routing:

- The resource set consists of all routing nodes $\mathcal{V}$ in the routing graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$.
- The consumer set consists of all nets $\mathcal{N}$ to be routed.
- The block of feasible solution for consumer $n \in \mathcal{N}$ is a convex hull $\mathcal{B}_n = \mathrm{conv}(\mathcal{B}_n^{\perp})$ of all feasible routing trees $\mathcal{B}_n^{\perp}$.
    - Every element $b_n \in \mathcal{B}_n$ has the form $b_n = \sum_{b^{\perp} \in B_n^{\perp}} x_{n,b^{\perp}} \cdot b^{\perp}$, i.e., a convex combination of elements in $\mathcal{B}_n^{\perp}$ with $x_{n,b^{\perp}} \geq 0$ and $\sum_{b^{\perp} \in B_n^{\perp}} x_{n,b^{\perp}} = 1$.
    - Because $b \in \mathcal{B}_n$ is continuous, "randomized rounding, rip–up and reroute" (see Section 2.4 in [GMN13]) is needed to get a solution from the dicrete set $\mathcal{B}_n^{\perp}$.
- The usage function $g_n(b_n) = \sum_{b^{\perp} \in B_n^{\perp}} x_{n,b^{\perp}} \cdot g_n(b^{\perp})$, where $(g_n(b^{\perp}))_v = 1$ when solution $b^{\perp}$ routes through routing node v, and otherwise, $(g_n(b^{\perp}))_v = 0$.
- And for a problem instance with feasible routing solutions, we know $\lambda^* = 1$ a priori.

10.3. Algorithm

> ($\sigma\text{-approximate}$ oracle in [MRV11], a.k.a. "block solvers") Given a constant $\sigma \geq 1$ and a consumer $n \in \mathcal{N}$, a $\sigma\text{-approximate}$ oracle $f_n : \mathbb{R}_+^{\mathcal{V}} \to \mathcal{B}_n$ takes a resource cost vector $y \in \mathbb{R}_+^{\mathcal{V}}$ as input and returns an element $f_n(y) \in \mathcal{B}_n$ such that $y^{\intercal} g_n(f_n(y)) \leq \sigma \cdot opt(y)$ where $opt(y) = \inf_{b \in \mathcal{B}_n} \{y^{\intercal} g_n(b)\}$.

Input:

- Resources $\mathcal{V}$ and consumers $\mathcal{N}$
- Usage functions g_n
- Oracle functions f_n
- Parameters $\epsilon > 0$ and $p_{\max} \in \mathbb{N}$

Output:

- (primal variables) Coefficients $\{x_{n,b^\perp}\}$ for $b_n = \sum_{b^\perp \in B_n^\perp} x_{n,b^\perp} \cdot b^\perp \in \mathcal{B}_n$
- ("dual" variables) Resource cost vector $y \in \mathbb{R}_+^{\mathcal{V}}$

Procedure:

```
1. Set $y := 1$ (i.e., $y_v := 1$ for $v \in \mathcal{V}$)
2. Set $x_{n,b^\perp} := 0$ for $n \in \mathcal{N}$ and $b^\perp \in \mathcal{B}_n^\perp$
3. for $p := 1$ to $p_{\max}$ do  // iterate for $p_{\max}$ phases
4.     for $n \in \mathcal{N}$ do
5.         Set $b^\perp := f_n(y)$  // Dijkstra's algorithm with node costs $y$
6.         Set $x_{n,b^\perp} := x_{n,b^\perp} + 1/p_{\max}$  // increase $x_{n,b^\perp}$'s weight by $1/p_{\max}$
7.         for $v \in \mathcal{V}$ with $(g_n(b^\perp))_v > 0$ do
8.             Set $y_v := y_v \cdot \exp(\epsilon \cdot (g_n(b^\perp))_v)$
9.         end for
10.    end for
11. end for
```

Details

- The "oracle" could be implemented as Dijkstra's algorithm in a cost-weighed graph.
    - Strictly speaking, the oracle functions should be an approximation algorithm for the Steiner tree Problem in Graphs (SPG).
- Parameters: $t = 125$ and $\epsilon = 1$ in [GMN13].
- The variables $\{x_{n,b^\perp} : n \in \mathcal{N}, b^\perp \in \mathcal{B}_n^\perp\}$ are not stored explicitly. Instead, we only maintain the tuples $\{(n, b^\perp, x_{n,b^\perp}) : x_{n,b^\perp} > 0\}$.
- The output coefficients $\sum_{b^\perp \in \mathcal{B}_n} x_{n,b^\perp} = 1$ can be interpreted as probabilities.
    - Randomized rounding samples routing tree $b_n^\perp$ for net n with a probability of $x_{n,b_n^\perp}$.
- The output resource cost vector $y_v = \prod_{n \in \mathcal{N}} \prod_{b^\perp \in \mathcal{B}_n} \exp(\epsilon \cdot x_{n,b^\perp} \cdot p_{\max} \cdot (g_n(b^\perp))_v)$.

- If there are conflicts $(\text{i.e.}, \exists v \in \mathcal{V}, \sum_{n \in \mathcal{N}} (g_n(b_n^{\perp}))_v > 1)$ after randomized rounding, we resort to resampling (<10% nets in [GMN13]), followed by rip–up and reroute (<5% runtime in [GMN13]).

10.4. Complexity

Let θ denote the time for a $\sigma\text{-approximate}$ oracle call.

> ("**Theorem 8**" in [MRV11]). We can compute a $\sigma(1{+}\omega)\text{-approximate}$ solution in time $O\left(\theta \cdot \log|\mathcal{V}| \cdot \left((|\mathcal{N}| + |\mathcal{V}|) \cdot \log\log|\mathcal{V}| + (\mathcal{N}| + |\mathcal{V}|\sigma)\sigma\omega^{-2}\right)\right).$

> ("**Main Theorem**" in [Bla22]). We can compute a $\sigma(1{+}\omega)\text{-approximate}$ solution in time $O(\theta \cdot (|\mathcal{N}| + |\mathcal{V}|) \cdot \log|\mathcal{V}| \cdot \omega^{-2}).$

> ("**Corollary 11**" in [MRV11]). We can compute a $\sigma(1{+}\omega)\text{-approximate}$ solution in time $O(\theta \cdot |\mathcal{N}| \cdot \log|\mathcal{V}| \cdot (\log|\mathcal{V}| + \sigma\omega^{-2})).$

The last improved bound is due to the fact that we can restrict $\mathcal{B}_n$ to $\{b \in \mathcal{B}_n : g_n(b) \leq \mathbf{1}\}$ without changing the optimum for routing as MMRSP.

10.5. Practical Implementation

All the (provable) tricks are from [MRV11].

10.5.1. Early termination

Let $\epsilon' = (\exp(\epsilon) - 1) \cdot \sigma,$ we can stop the core algorithm when $\mathbf{1}^{\mathsf{T}} y > |\mathcal{V}| \cdot \exp(p \cdot \epsilon'/(1 - \epsilon'))$ in phase $p,$ since we are often interested in a solution with $\lambda^* \leq 1.$

10.5.2. Solution reuse

We only need to call the $\sigma\text{-approximate}$ oracle at the first iteration or when the saved solution is not good enough, according to a parameter $\tau \geq 1.$ Specifically, solution b_n is not good enough when $y^{\mathsf{T}} g_n(b_n) > \tau \cdot y_n^{\mathsf{T}} g_n(b_n).$ Here y is the current node cost vector, and y_n was the node cost vector when we call the oracle last time to obtain $b_n.$

This implementation behaves as a $\sigma\tau\text{-approximate}$ oracle.

10.5.3. Parallelization

The details of parallelization can be found in "Section 6. Parallelization" in [MRV11].

> ("Volatility–Tolerent Oracle" in [MRV11]). $\mathcal{F}_n : (\mathcal{V} \to \mathbb{R}_+) \to \mathcal{B}_n \times \mathbb{R}_+$ is procedure that takes the resource cost querier $\mathcal{Q} : \mathcal{V} \to \mathbb{R}_+$ as input, instead of a resource cost vector $y \in \mathbb{R}_+^{\mathcal{V}}$, and produces a pair $(b, z) \in \mathcal{B}_+ \times \mathbb{R}_+$ as output.
>
> $\mathcal{F}_n$ is a volatility–tolerent oracle if for any call to it the following condition is satisfied: For $v \in \mathcal{V}$ let Y_v be the set of resource costs sampled by queryign $\mathcal{Q}(v)$ in this execution of $\mathcal{F}_n$. Then there is a $y'_v \in Y_v$ for each $v \in \mathcal{V}$ with $Y_v \neq \emptyset$ such that for each cost vector $y \in \mathbb{R}_+^{\mathcal{V}}$ with $y_v = y'_v$ for $v \in \mathcal{V}$ and $Y_r \neq \emptyset$, we have $z = y^\intercal g_n(b)$ and $z \leq \sigma \cdot opt_n(y)$.

> Trivially, volatility–tolerant block solvers can be implemented by sampling the price of each resource once and then calling a standard block solver which gets the resulting static vector as input.

Dijkstra’s shortest path algorithm is such an example that the cost of each node is queried at most once, i.e., $|Y_v| \leq 1$ for $v \in \mathcal{V}$.

Input:

- Resources $\mathcal{V}$ and consumers $\mathcal{N}$
- Usage functions g_n
- Oracle functions $\mathcal{F}_n$ with Querier $\mathcal{Q}_\mu$
    - $\mathcal{Q}_\mu(v) = \exp(\epsilon \cdot \mu_v)$ for $\mu \in \mathbb{R}_+^{\mathcal{V}}$ and $v \in \mathcal{V}$
- Parameters $\tau \geq 1, \quad \epsilon > 0,$ and $p_{\max} \in \mathbb{N}$

Output:

- Coefficients $\{x_{n,b^\perp}\}$ for $b_n = \sum_{b^\perp \in B_n^\perp} x_{n,b^\perp} \cdot b^\perp \in \mathcal{B}_n$

[MRV11] Procedure $(\text{parapllel in } n \in \mathcal{N})$:

```
1. (globally) Set $\mu := 0, \quad \mu' := 0, \quad t := 0, \quad X_n := 0, \quad \bar{X}_n := 0$
2. for $p := 1 \text{ to } p_{\max}$ do  // iterate for $p_{\max}$ phases
3.     (locally) Set $(b, z) := \mathcal{F}_n(\mathcal{Q}_\mu)$
4.     (atomically) Set $\mu' := \mu' + g_n(b)$
5.     memory_barrier
6.     (atomically) Set $t := t + 1$  // time stamp
7.     memory_barrier
8.     (locally) Set $y_v := \mathcal{Q}_{\mu'}(v)$ for $v \in \mathcal{V}$ with $(g_n(b))_v > 0$
9.     memory_barrier
10.    (locally) Set $y_v := y_v \cdot \mathcal{Q}_\mu(v) \cdot \exp(-\epsilon \cdot u_v)$ for $v \in \mathcal{V}$ with $(g_n(b))_v > 0$
11.    if $y^\intercal g_n(b) \leq \tau \cdot z$ then
12.        (locally) Set $X_n := X_n + 1$  // accept solution
13.        (locally) Set $x_{n,b} := x_{n,b} + 1/p_{\max}$
14.        (atomically) Set $\mu := \mu + g_n(b)$
15.        memory_barrier
16.    else
17.        (locally) Set $\bar{X}_n := \bar{X}_n + 1$  // reject solution
18.    end if
19.    (atomically) Set $\mu' := \mu' - g_n(b)$
20. end for
21. // sequentially repeat the $\bar{X}_n$ rejected attempts
```

("Lemma 15" in [MRV11]) In each accepted solution, $\mathcal{F}_n$ acts as a $\sigma\tau\text{-approximate}$ oracle.

11. A* Algorithm

> Rios, Luis Henrique Oliveira, and Luiz Chaimowicz. "Pnba*: A parallel bidirectional heuristic search algorithm." *ENIA VIII Encontro Nacional de Inteligê ncia Artificial*. 2011.

$d(x, y)$: The weight/cost associated with edge (x, y)

$d^*(x, y)$: The cost of the shortest path from x to y

$h(x)$: The heuristic function

$h^*(x)$: $h^*(x) = d^*(x, x_{goal})$

> In the context of A*, the heuristic function may have two important properties. It is said to be **admissible** if $h(x) \le h^*(x)$ for all nodes x , i.e., the heuristic function never overestimates the real cost. Also, the heuristic function is said to be **consistent** or **monotone** if $h(x) \le d^*(x, y) + h(y)$ for any nodes x, y (or, alternatively, $h(x) \le d(x, y) + h(y)$ for any edge (x, y)). This can be viewed as a kind of triangle inequality: each side of a triangle can not be larger than the sum of the two other. Every consistent heuristic function is also admissible. The importance of this distinction will be explained below.
>
> A* is complete, i.e, it always terminates with a solution wherever one exists, if the following conditions are satisfied: (i) nodes must have a finite number of successors and (ii) the weight associated with the edges must be positive. Also, it is admissible (returns the lowest cost solution) if one more condition is satisfied: heuristic admissibility. Imposing a more strict restriction (consistency) on the heuristic function allows a simpler and more efficient version of A*, in which a node needs to be expanded at most once. From now on, the heuristic function is assumed to be consistent.

admissible: $h(x) \le h^*(x)$ $\rightarrow$ The A* algorithm returns the lowest cost solution

consistent: $h(x) \le d(x, y) + h(y)$ $\rightarrow$ Every node needs to be expanded at most once