Generic pathfinding

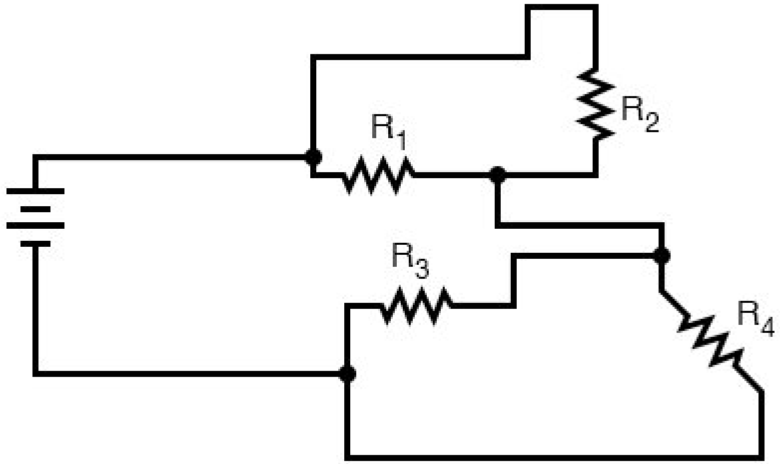
boost::graph for dummies

No raw loops

- If you have a loop in the middle of a function
    - It probably shouldn’t be there
- STL provides many commonly used algorithms
- Algorithms often work with iterators.
    - STL provides many one dimension containers with **begin** and **end**
    - Pointer is also an iterator

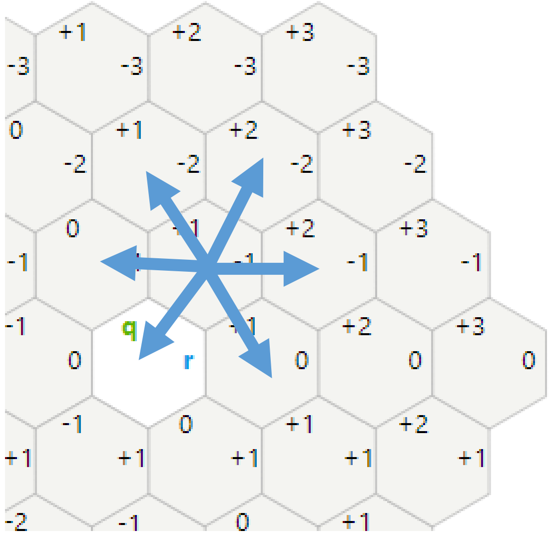
Graphs

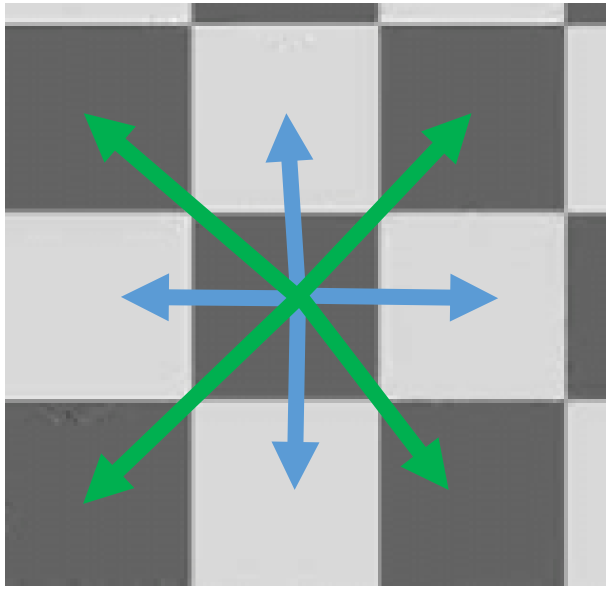
- A set of vertices (nodes, locations, junctions)
- A set edges (connections, links) each is a pairs of members of first set
- There are many ways to define sets in code
- Edges and vertices may have properties



Graphs - grid


Rapid Healing In Progress
50
'Ether'
Yong Sun Shin
TARGET PREVIEW:
END TURN
SHOW CHARACTER DETAILS
T-4
R2


+1 +2 +3
-3 -3 -3 -3
0 +1 +2 +3
-2 -2 -2 -2
0 +2 +3
-1 -1 -1
-1 +2 +3
0 0 0 0
q
r
-1 0 +1 +2
+1 +1 +1 +1 +1
-2 -1 0 +1

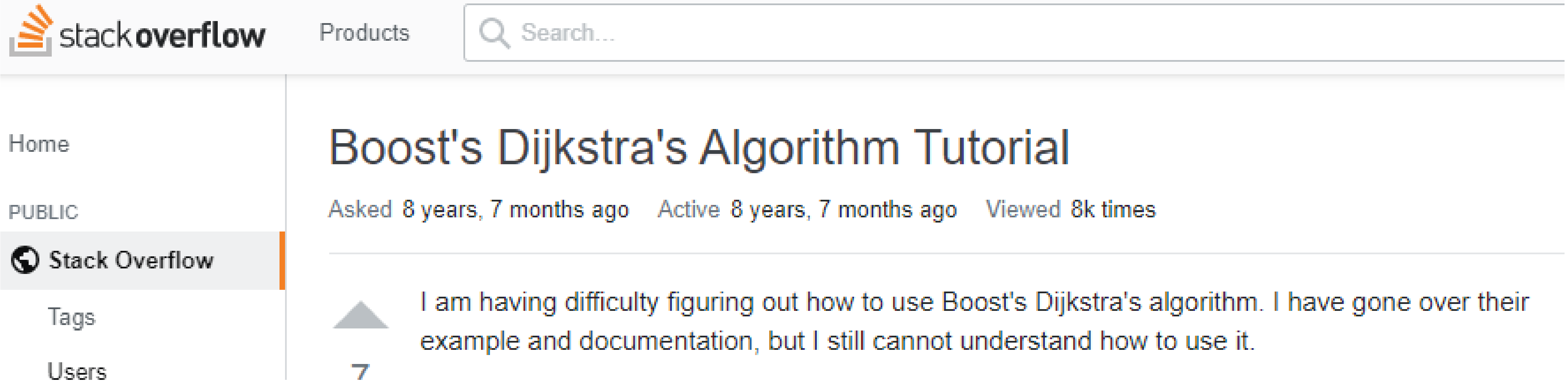


Pathfinding

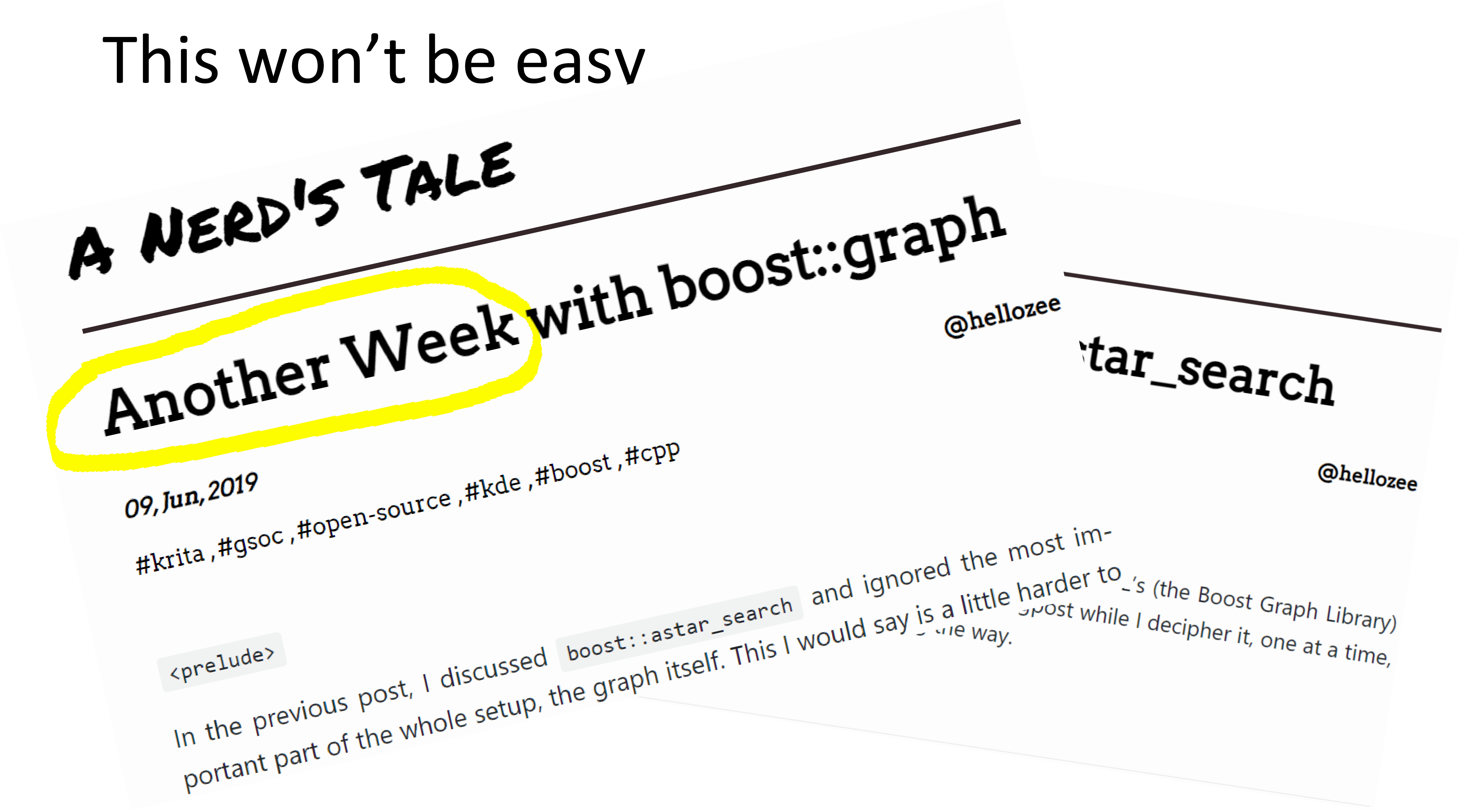
- Find best path from point to
  - Another point
  - All points within given range
- A set of well known algorithms
  - BFS, Dijkstra, A*
- Can be applied to (almost) any graph
  - And even infinite graph-like structure

There is no A* in STL

- Let’s Google generic pathfinding in C++
  - We have boost::graph (since boost 1.18!)
- Does boost::astar_search really have 11 parameters? ;)
  - Let’s decipher it



This won’t be easy



The input

- Begin vertex
- End vertex or some other termination condition (optional)
  - For example we want to see all spots reachable within fixed time
  - Perhaps we want to find path to 5 different spots, in the same general direction
  - Anyway it is vastly different from begin and end of array
- Way to get a list of points adjacent to a given point

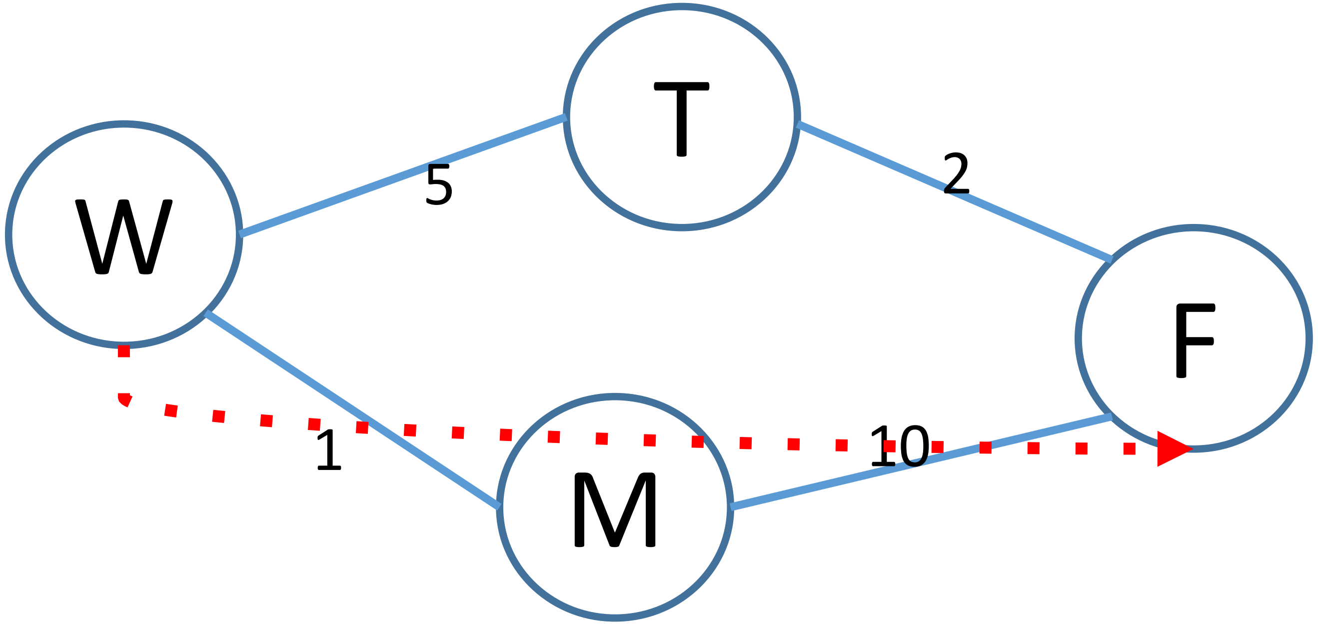
The input

- The full list of vertices may be available in advance or generated dynamically
- Travel cost from point to adjacent point (weight)
  - The generic way to represent it should be some kind of callable
  - For BFS it is constant
- For A* estimated cost from point to end

The output

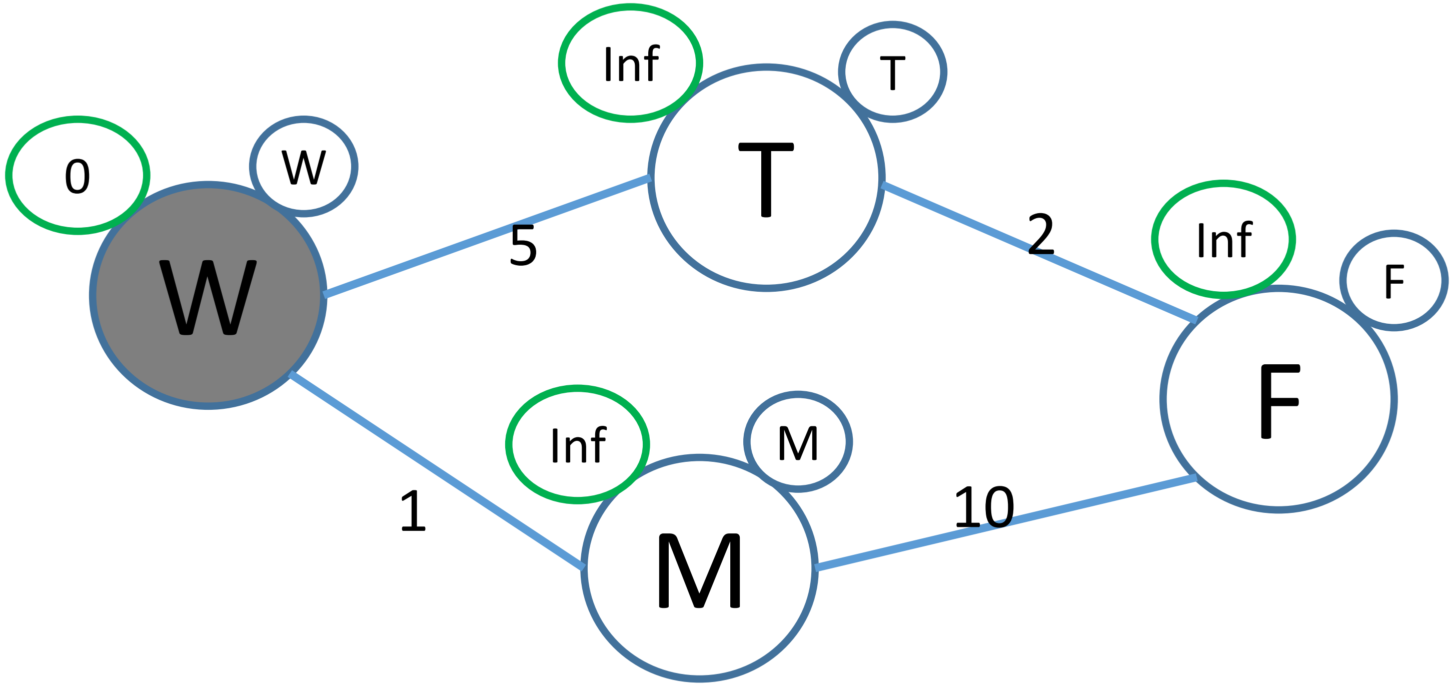
- The simplest exercise case is just the distance between two points.
- More practical case is distance and path.
    - The topology of output is the same as input!

The Dijkstra algorithm



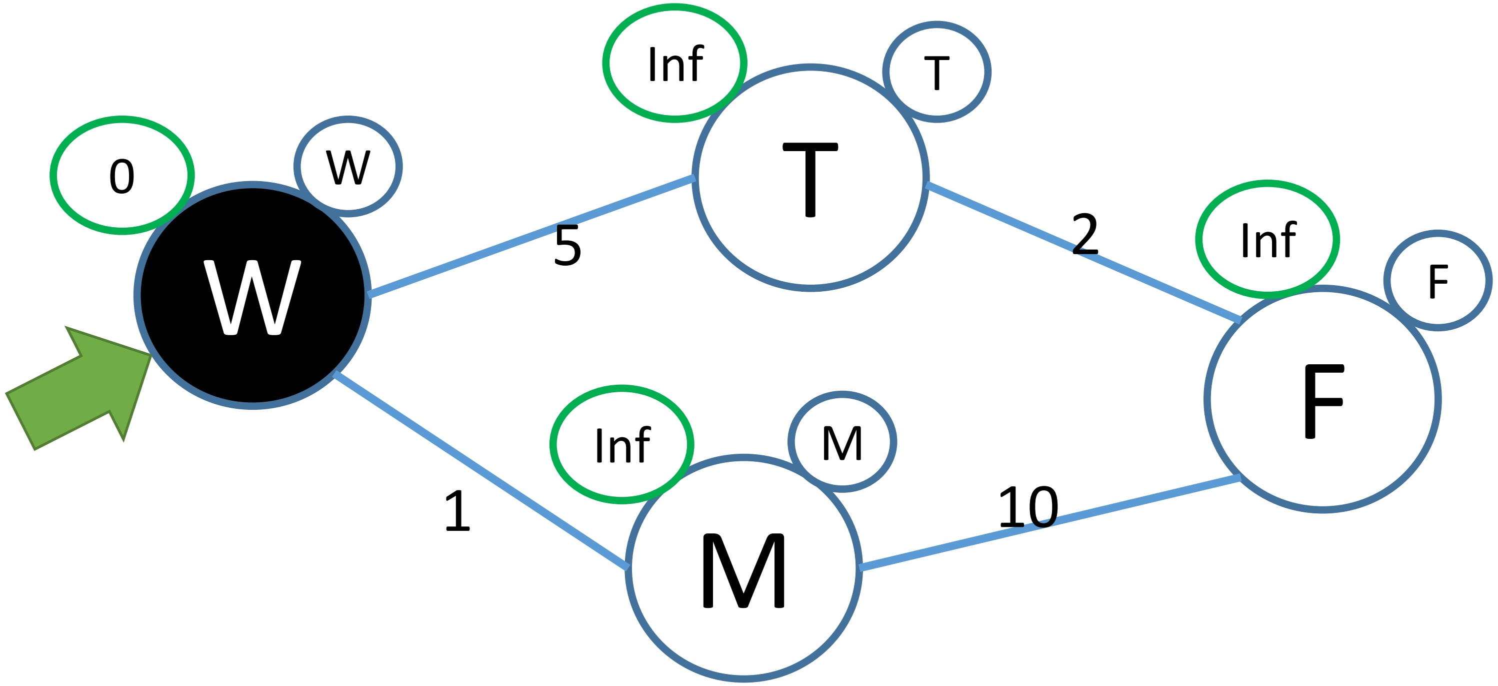
??????

The Dijkstra algorithm



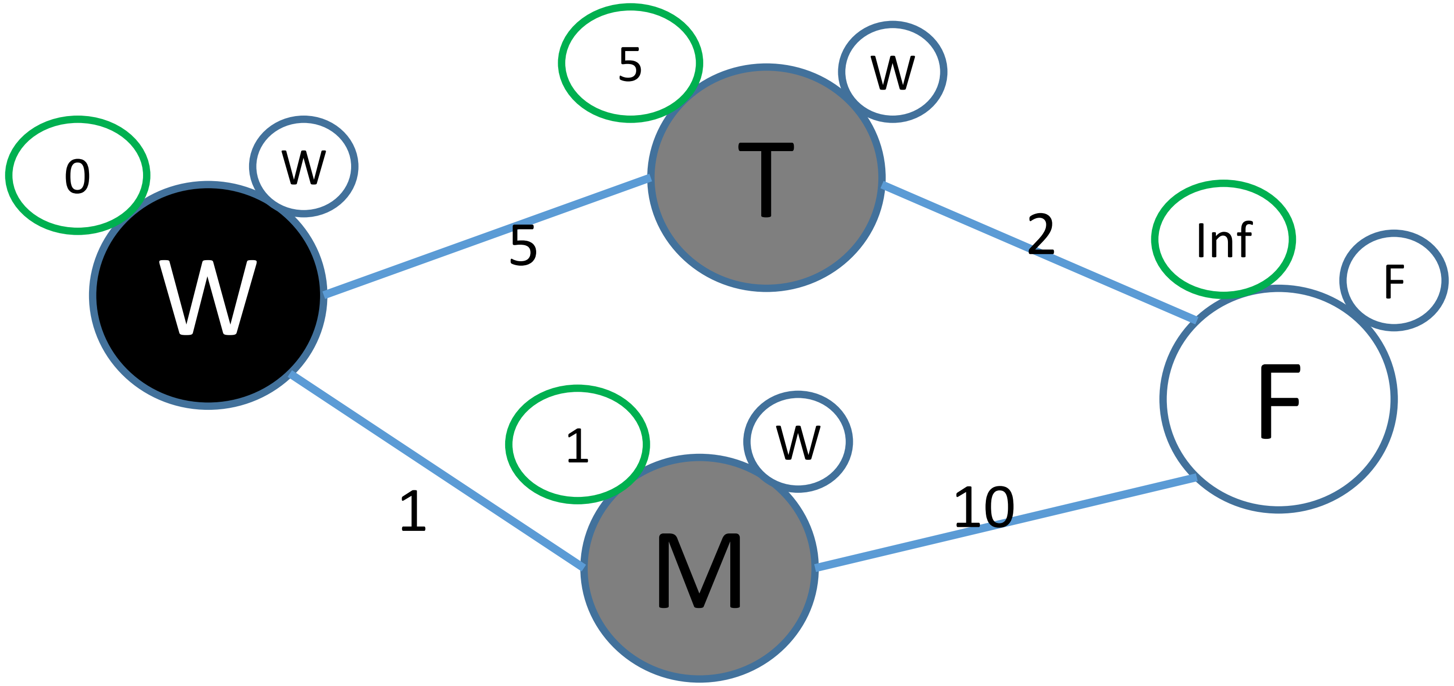
Priority	Node
0	W

The Dijkstra algorithm



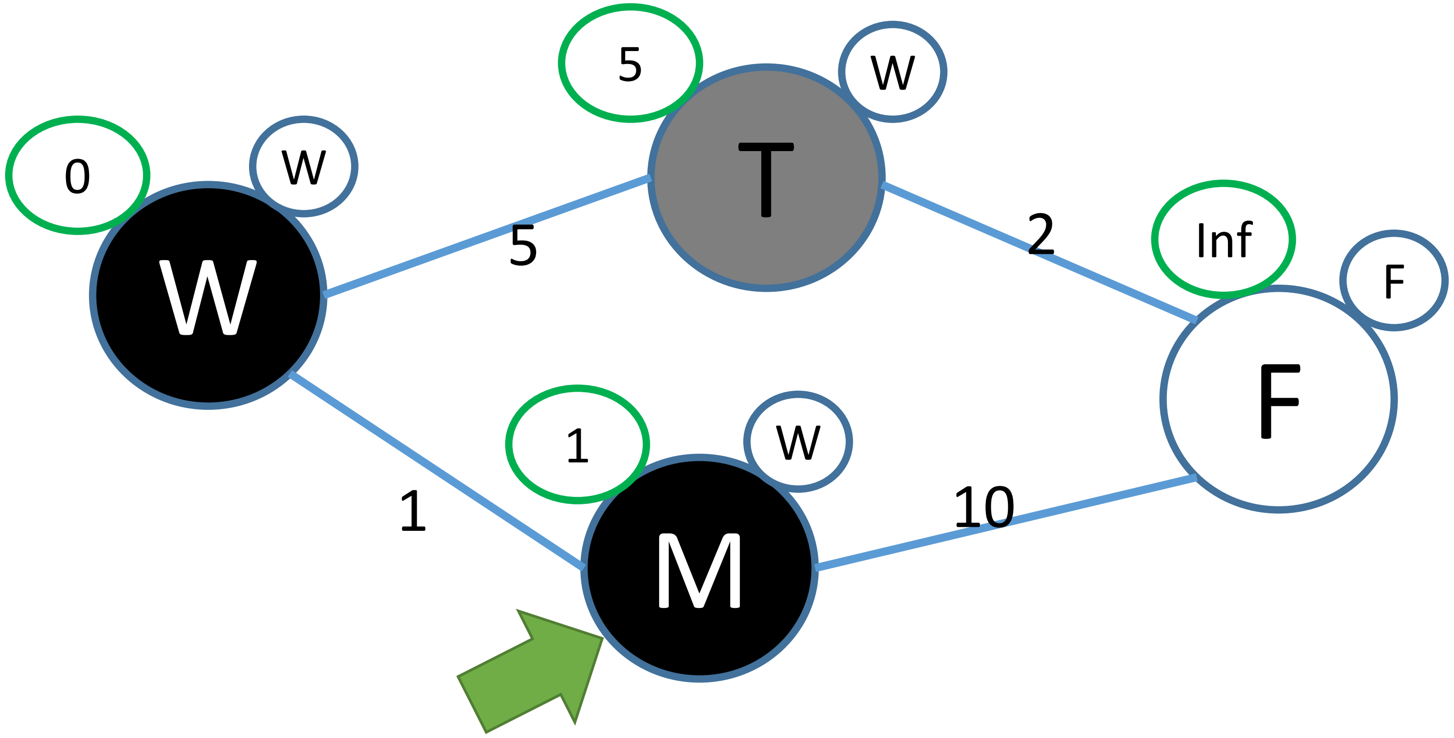
Priority	Node

The Dijkstra algorithm



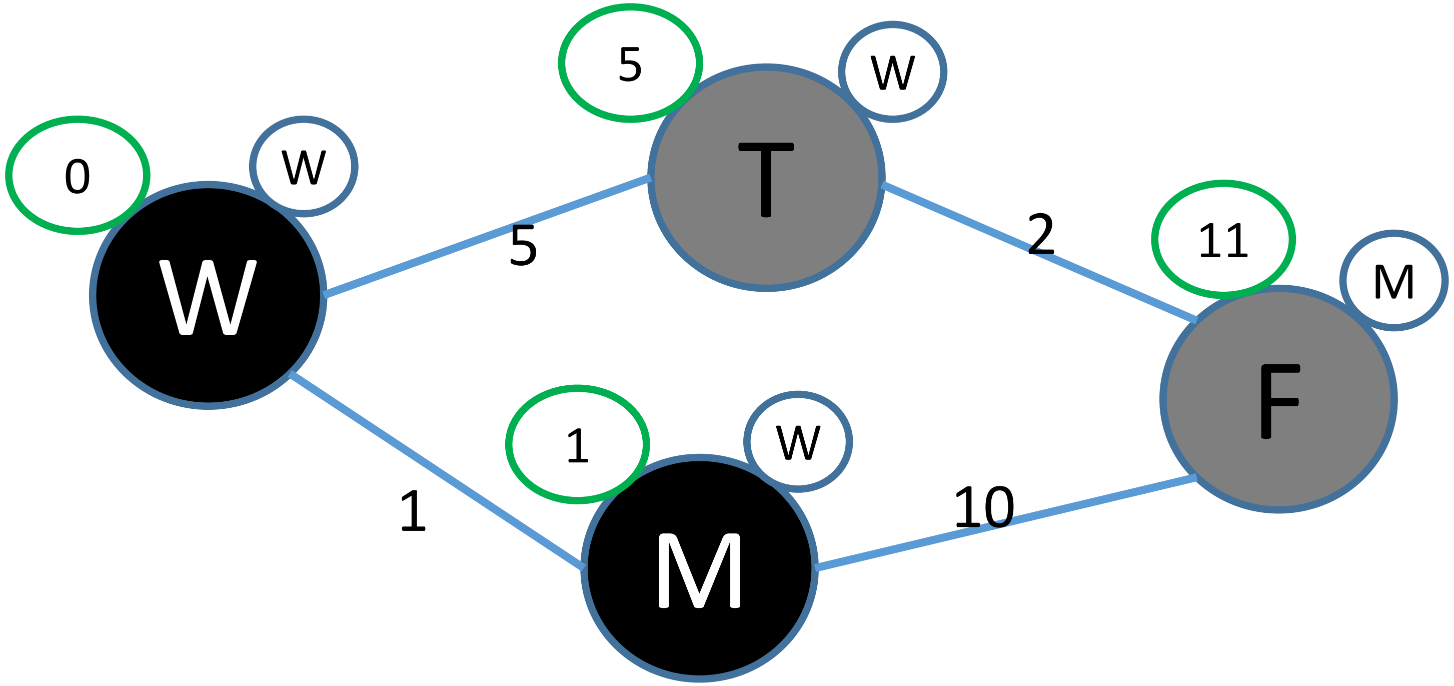
Priority	Node
1	M
5	T

The Dijkstra algorithm



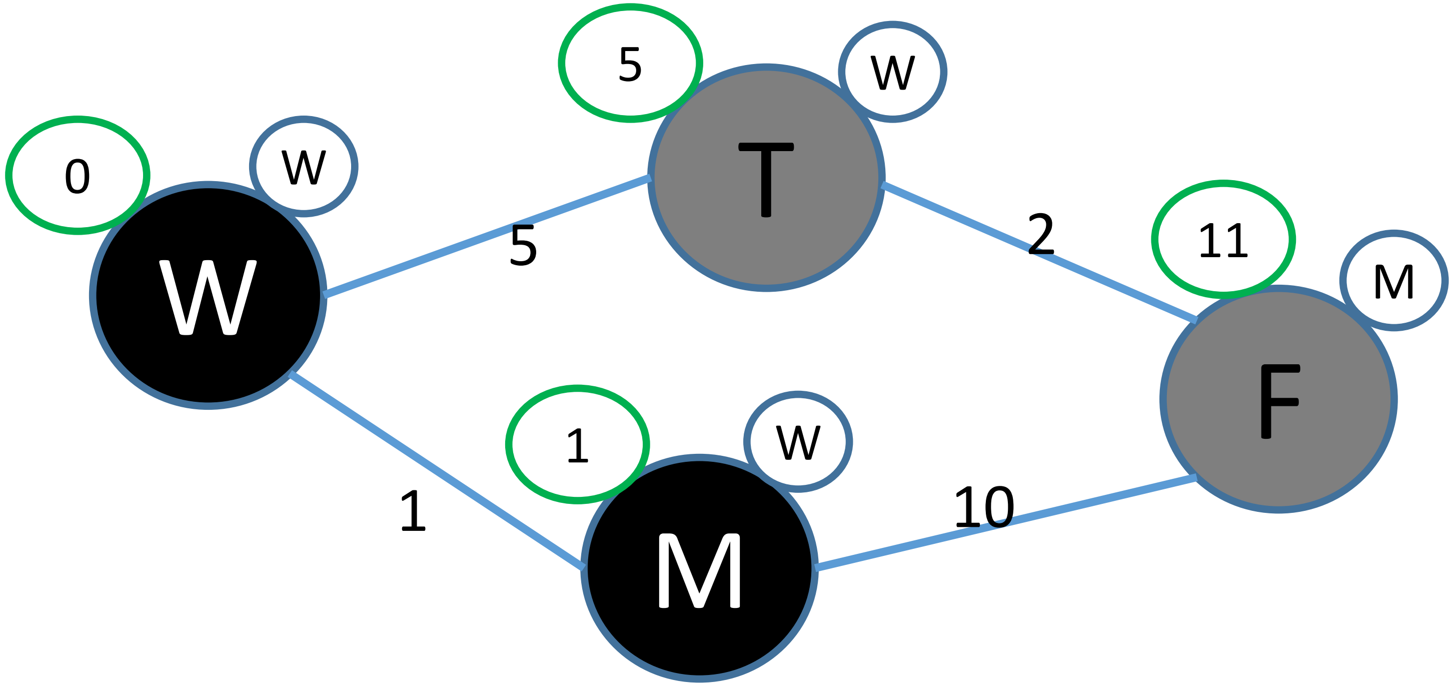
Priority	Node
5	T

The Dijkstra algorithm



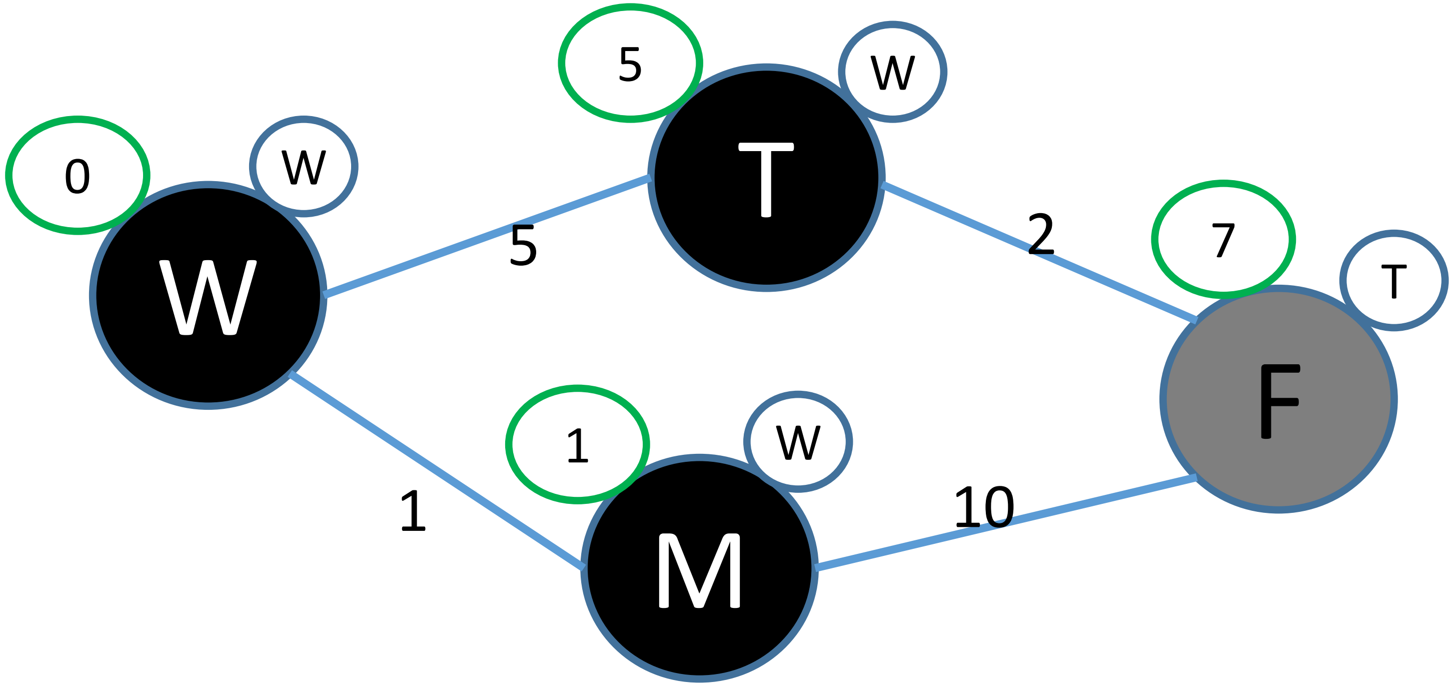
Priority	Node
5	T
11	F

The Dijkstra algorithm



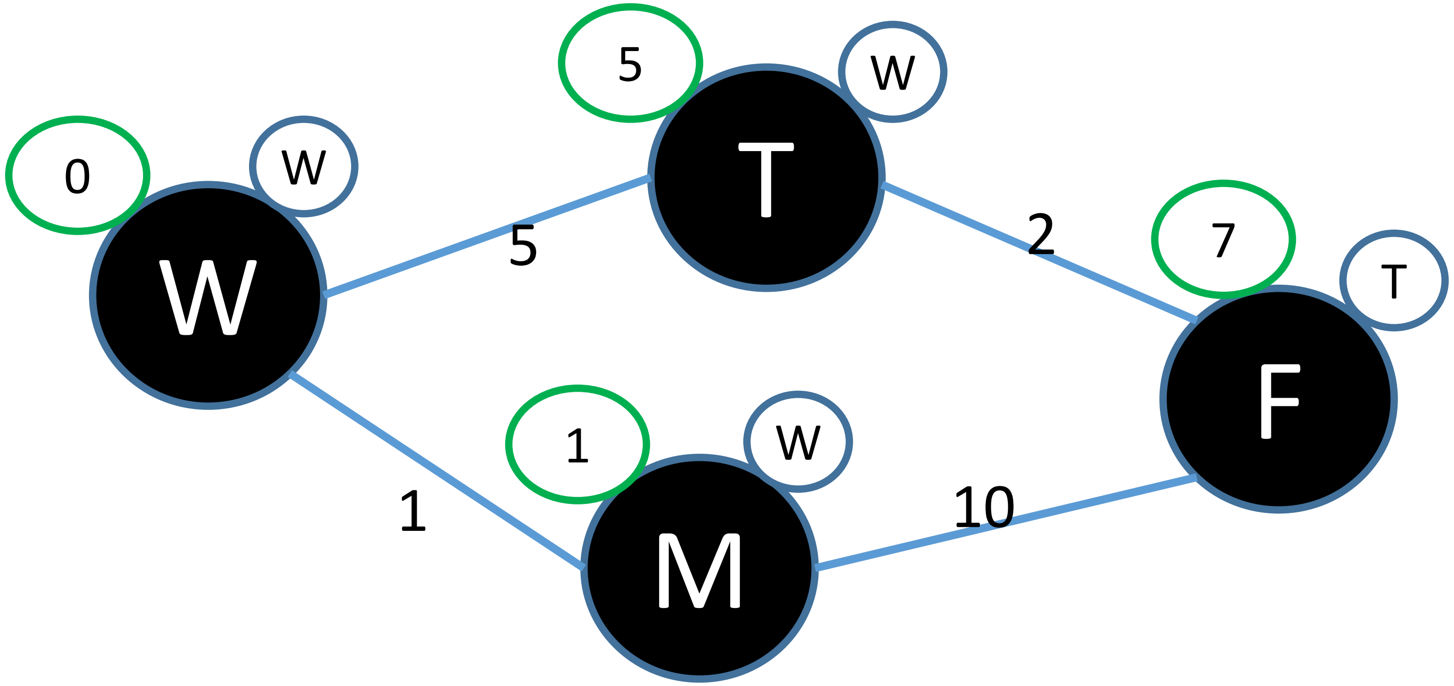
Priority	Node
5	T
11	F

The Dijkstra algorithm



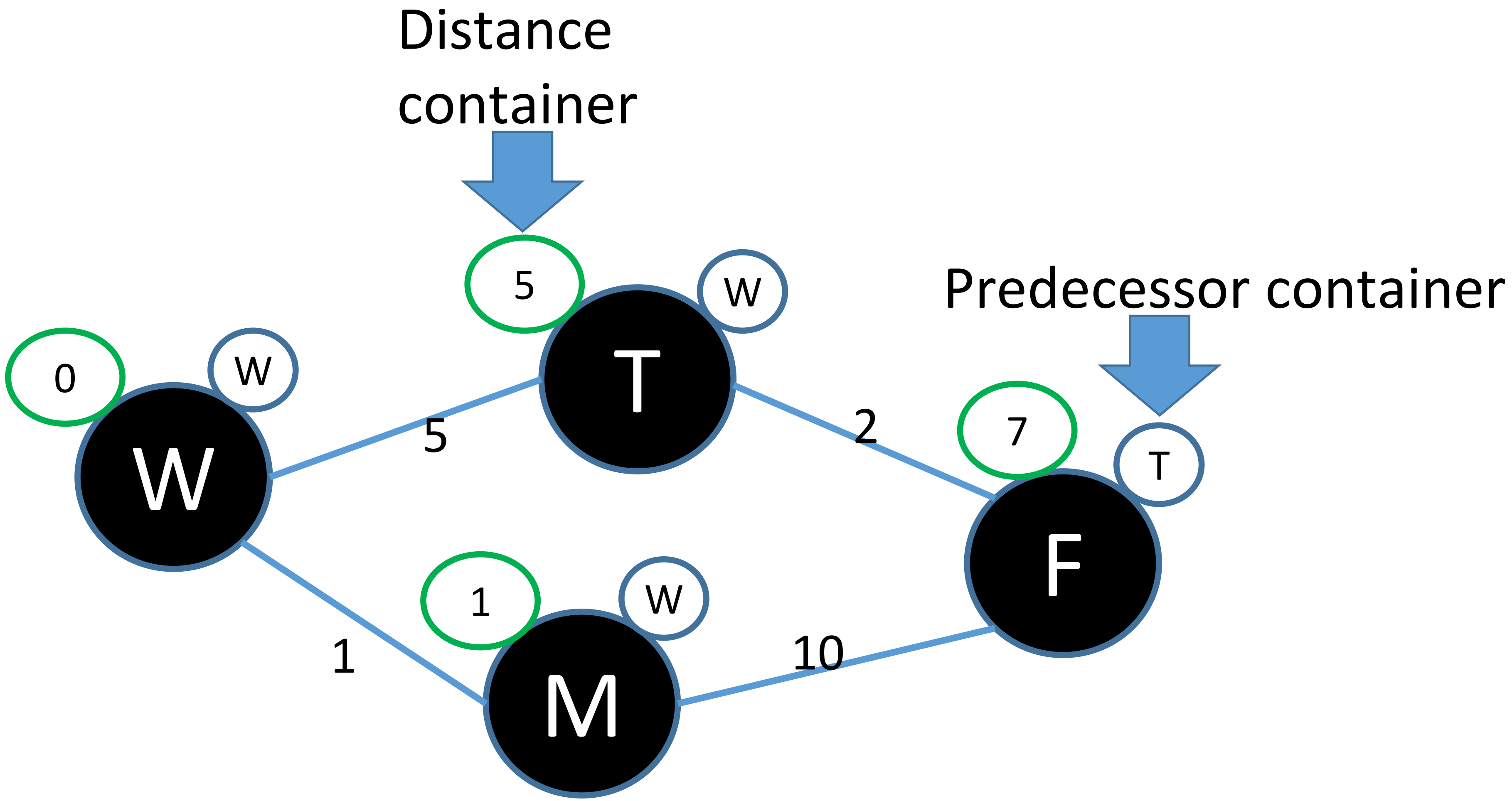
Priority	Node
7	F
~~11~~	~~F~~

The Dijkstra algorithm



Priority	Node

The Dijkstra algorithm outputs



Key points

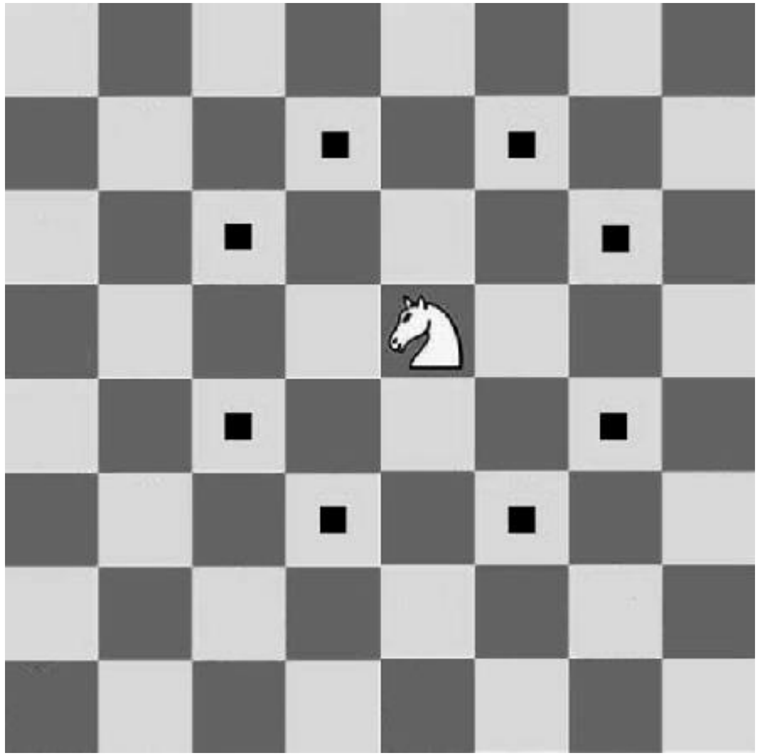
- It is somewhat similar to std::transform. Input is likely a container and output is same size as input.
- Unlike std::transform the input is not one dimensional. operator++ is not enough!
- std::transform at every step of execution knows one variable place in input container and **matching** place in output container

Indices vs iterators

- In case of STL’s std::transform if you have two iterators pointing at matching places doing ++ or advance(X) on both will keep them matching.
- Pathfinding at every step knows multiple variable locations in original data structure

Let’s start with simple tasks to test boost::graph usability

- Given an empty chess board of size NxN
  1. Find minimum number of knight moves K to get from {X1, Y1} to {X2, Y2}
     - Should be super easy to implement manually
  2. Find a set S of fields reachable from {X, Y} within M moves
     - Not much harder than first
  3. Provide list of unreachable squares



Graphs in boost::graph

- Graph is a concept
  - A set of valid operations on something used as template argument
- Base graph requires very basic stuff for example mostly typedefs
- Other graphs require supporting more operations
  - IncidenceGraph must provide operations for traversing neighbors of vertex
  - VertexListGraph must provide number and iterable list of all vertices

Property maps

- Concept of a universal container
- The interface for property maps consists of three functions:
  - `get()`
  - `put()`
  - `operator[]`
- Read only container can be based on calculation, boost provides some utilities to help for implementing it.

Boost – A*

```
template <typename VertexListGraph, typename AStarHeuristic,
   typename AStarVisitor, typename PredecessorMap,
   typename CostMap, typename DistanceMap, typename WeightMap,
   typename VertexIndexMap,
   typename ColorMap, typename CompareFunction, typename CombineFunction,
   typename CostInf, typename CostZero>
inline void
astar_search
  (const VertexListGraph &g, typename
graph_traits<VertexListGraph>::vertex_descriptor s,
   AStarHeuristic h, AStarVisitor vis, PredecessorMap predecessor, CostMap
cost,
   DistanceMap distance, WeightMap weight, VertexIndexMap index_map,
ColorMap color,
   CompareFunction compare, CombineFunction combine, CostInf inf, CostZero
zero);
```

Dijkstra should be simpler

```
template <typename Graph, typename DijkstraVisitor,
  typename PredecessorMap, typename DistanceMap,
  typename WeightMap, typename VertexIndexMap,
  typename CompareFunction, typename CombineFunction,
  typename DistInf, typename DistZero, typename ColorMap =
default>
void dijkstra_shortest_paths
  (const Graph& g,
   typename graph_traits<Graph>::vertex_descriptor s,
   PredecessorMap predecessor, DistanceMap distance,
   WeightMap weight, VertexIndexMap index_map,
   CompareFunction compare, CombineFunction combine,
   DistInf inf, DistZero zero,
   DijkstraVisitor vis, ColorMap color = default)
```

BFS is even simpler

```
template <class Graph, class Buffer,
  class BFSVisitor, class ColorMap>
void breadth_first_search(const Graph& g,
 typename graph_traits<Graph>::vertex_descriptor s,
 Buffer& Q, BFSVisitor vis, ColorMap color);
```

Looks too simple to be what we are looking for, just a building block for Dijkstra and other similar algorithms

OK, let’s decipher Dijkstra arguments

- `const Graph& g`
  - Provides a way (iterator pair) to get the list of neighbours of given vertex. Defines the actual topology.
  - Also full list of vertices in graph
- `graph_traits<Graph>::vertex_descriptor s`
  - Start point for search. There are some overloads with multiple starting points

OK, let’s decipher Dijkstra arguments

- `PredecessorMap predecessor`
  - Main output. The actual path found. The previous position.
- `DistanceMap distance`
  - Also output. The distance to a given point.

OK, let's decipher Dijkstra arguments

- `WeightMap weight`
  - Input. The cost to move between adjacent nodes.
- `VertexIndexMap index_map`
  - Translate coordinates to single number. Why is it a must?
- `CompareFunction, CombineFunction, DistInf, DistZero`
  - I hope there is an overload to provide a good default

OK, let’s decipher Dijkstra arguments

- `DijkstraVisitor vis`
  - Observes the search process.
- `ColorMap color = default`
  - Whether the node was visited, finalized…

Grid 2D

- So we have boost::grid_graph<N> and it can be used for grids like . It will provide the required functions
  - to count vertices
  - … and iterate over all vertices
  - … and a function to get all adjacent nodes
  - … and mapping indices to coordinates, coordinates to indices
- By default adjacency is horizontal and vertical.
- Can we have it use knight's move adjacency instead of default?
  - Because the example on the next slide won't pass the first unit test

Something that compiles

```
using gg2d = boost::grid_graph<2, int>;
gg2d board(dimensions);
dijkstra_shortest_paths(board, begin,
    p_map, dmap, weight,
    boost::grid_graph_index_map<
    gg2d,
    typename gg2d::vertex_descriptor,
    typename gg2d::vertices_size_type>(board),
    std::less<int>(),
    boost::closed_plus<int>((std::numeric_limits<int>::ma
    (std::numeric_limits<int>::max)(),
    0,
    boost::make_dijkstra_visitor(boost::null_visitor()));
```

About boost parameter library

- There is an overload that uses boost parameter library.
  - Imitate named function parameters
  - Give reasonable default to parameters from the middle of the list
- Did not work for me
  - Worked on original grid_graph, but not on subclass

No simple knight’s move

- grid_graph uses transform_iterator
- The transform function is part of the class, not customizable
- The actual code is for orthogonal N dimensional grid and the iteration logic is part of grid_graph all about supporting any number of dimensions
- Let’s create our own
    - Do something wrong and you get compilation error with so much templates you will never understand it

Let’s try to extend grid_2d

```
struct my_grid : public boost::grid_graph<2,
int>
{
  my_grid(vertex_descriptor dims) :
  boost::grid_graph<2, int>(dims) {}
};
```

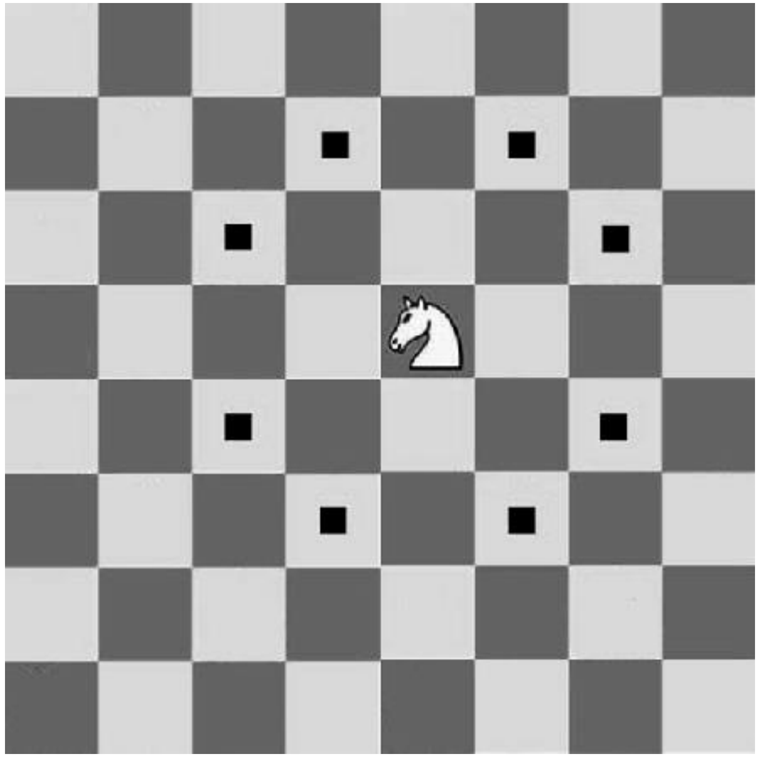
Now what?

- Let’s make it generic so it accepts a list to iterator over valid move range
  - We could use it for hex grid!
- Let’s add iterator the naïve way

Some snippets

```
int minKnightMovesEx(int n, CoordT begin, CoordT end) {
    std::array<CoordT, 8 > moves = {
        CoordT{ 1, 2 },
        CoordT{ 1, -2 },
        CoordT{ -1, 2 },
        CoordT{ -1, -2 },
        CoordT{ 2, 1 },
        CoordT{ 2, -1 },
        CoordT{ -2, 1 },
        CoordT{ -2, -1},
    };

    CoordT dimensions{n, n};
    using my_graph_t = my_graph<int, decltype(moves.begin())>;
    my_graph_t board(moves.begin(), moves.end(), dimensions);
```



Some snippets

```
template <typename IndexType, class IterType>
struct my_graph : public boost::grid_graph<2,
IndexType>
{
   typedef my_graph type;
   my_graph(IterType moves_begin, IterType moves_end,
            vertex_descriptor dims) :
            boost::grid_graph<2, int>(dims), m_dims(dims),
            m_moves_begin(moves_begin),
            m_moves_end(moves_end) {}
```

Some snippets

```
friend inline std::pair<typename type::out_edge_iterator,
typename type::out_edge_iterator>
out_edges(typename type::vertex_descriptor vertex,
const type& graph)
{
    return std::make_pair(out_edge_iterator(graph.m_moves_begin,
    graph.m_moves_end, graph.m_dims, vertex),
    out_edge_iterator(graph.m_moves_end, graph.m_moves_end,
    graph.m_dims, vertex));
}
```

Some snippets

```
friend inline degree_size_type
    out_degree(typename type::vertex_descriptor vertex,
        const type& graph)
{
    throw std::logic_error("my_graph does not support out_degree");
}
```

- Incidence graph concept requires this function.
- Dijkstra does not use it
- And we do not want to implement it, but must make sure the one from grid_graph is not used.

Some snippets

```
struct out_edge_iterator :
public boost::forward_iterator_helper<out_edge_iterator,
std::pair<vertex_descriptor, vertex_descriptor> >
{
out_edge_iterator() = default;
out_edge_iterator& operator=(const out_edge_iterator& other) = default;
out_edge_iterator(IterType begin, IterType end,
vertex_descriptor dims,
const vertex_descriptor& vertex)
:m_current(begin), m_end(end), m_dims(dims)
{
m_edge.first = vertex;
update_and_skip_out_of_bounds();
}
```

Some snippets

```
bool valid_coordinate(const vertex_descriptor& coord) {
    return (coord[0] < m_dims[0]) && (coord[1] < m_dims[1])
        && (coord[0] >= 0) && (coord[1] >= 0);
}

void update_and_skip_out_of_bounds() {
    while (m_current != m_end)  {
        m_edge.second[0] = m_edge.first[0] + (*m_current)[0];
        m_edge.second[1] = m_edge.first[1] + (*m_current)[1];
        if (!valid_coordinate(m_edge.second)) {
            ++m_current;
        } else {
            return;
        }

    }
}
```

Let’s add some stuff

- Perhaps we want a way to mark certain cells as unreachable
    - Extract coordinate validation function to parameter
- Terminate upon reaching certain distance.
- Perhaps we want a way to mark certain cells as expensive to move into
    - Just use a different weight map

Quick and dirty cell validator

```
template <typename IndexType>
struct bounds_validator
{
    typedef boost::array<IndexType, 2> vertex_descriptor;
    bounds_validator() = default;

    bool operator()(const vertex_descriptor& coord)
    {
        return (coord[0] < m_dims[0]) && (coord[1] < m_dims[1])
            && (coord[0] >= 0) && (coord[1] >= 0);
    }

    bounds_validator(vertex_descriptor dims) : m_dims(dims) {}

    vertex_descriptor m_dims;
};
```

Visitor

- Has methods called on certain events during search. For example:
  - **vis.discover_vertex(u, g)** is invoked the first time the algorithm encounters vertex u.
- Can serve to collect output
  - For example if you need to build a list of locations reachable withi   n time X and another list for locations reachable between X+1 to 2X

Visitor

- Has methods called on certain events during search. For example:
  - **vis.discover_vertex(u, g)** is invoked the first time the algorithm encounters vertex u.
- Can serve to collect output
- For example if you need to build a list of locations reachable within time X and another list for locations reachable between X+1 to 2X
  - Visitor can ask to be called when path was found and check how the result in distance map compares to X.
  - Either store the vertex in one of two result lists or exit

Let's look at examples in boost documentation

```
struct astar_goal_visitor : public boost::default_astar_visitor
{
    astar_goal_visitor(vertex_descriptor goal) : m_goal(goal) {};

    void examine_vertex(vertex_descriptor u, const
filtered_grid&)
    {
        if (u == m_goal)
            throw found_goal();
    }

private:
    vertex_descriptor m_goal;
};
```

- Holy flipping shine! Does boost documentation actually recommend using exceptions for normal non-exceptional flow control?

Alternatives

- “LEMON” Graph Library
  - Often less annoying syntax
  - Different algorithm set
    - No A*
  - Weird iterators
- LEDA
  - Commercial
  - Bad documentation

std::graph

- Currently just a proposal.
    - Not even close to being approved.
- Defines many concepts boost like.
- No working implementation.
- Based on stronger language (C++20) than BGL (C++98) and should be easier to use.

BGL - The good

- Every underlying data structure can be adopted to work with BGL algorithms
- Some parts are implemented in simplest, given inherent problem complexity, way.
- Implements many different algorithms
- Some algorithms are easy to use
- Header only

BGL - The desired

- The documentation and examples are severely lacking.
- Overloads with less arguments for common needs are missing
  - Visitor for search for given node, up to certain distance, binning by distance range.
- Exceptions as the natural way to implement non exceptional control flow.
- Lots of minor stuff like IncidenceGraph required to provide a number of neighbors in advance.

Discussion

- It is easier to learn and use std::sort, than to write own sort for specific container and type
  - It is easier to learn and use std::rotate, than to write your own rotate for specific type
- For some problems like topological sort on adjacency list BGL is not harder to use, as to write your own.
- It is much easier to write your own specific case pathfinding than to learn and use the implementation by BGL
- std::graph authors should think twice about tradeoff between genericity and easy of use.

Recommended reading/watching

- Sean Parent – basically everything
  - No raw loops
  - No incidental data structures
- SG19 std::graph proposal
- ~~Online Boost Graph documentation~~

Thank you

- Questions?

muxecoid@gmail.com