Video Game Algorithms for Use in Robotics

April 16, 2011

Lucas Keyes
Gameplay Programmer, Griptonite Games

Purpose

- Present some programming concepts that I use as a video game developer that can be used for programming robots

Video Games vs Robots

Video Games – Characters or Avatars

- Need to appear intelligent
- React to the player and the environment
- Since the environment is simulated, can possibly know everything about it

Robotics – Robots

- Need to fulfill certain tasks
- React to the operator(s) and environment
- Cannot know everything about the environment, but can internally model it based off of different sensory input (touch, sonar, vision)

Finite State Machines

- Very popular way to imbue “intelligence” in game AI
- Definition
  - “A device, or model of a device, that has a finite number of states it can be in at a given time and can operate on input to either make transitions from one state to another or to cause an output or action to take place. A finite state machine can only be in one state at any moment in time.”

Advantages

- Quick and easy to code
- Easy to debug
- Little computational overhead
- Intuitive
- Flexible
- Easily adaptable to object-oriented programming

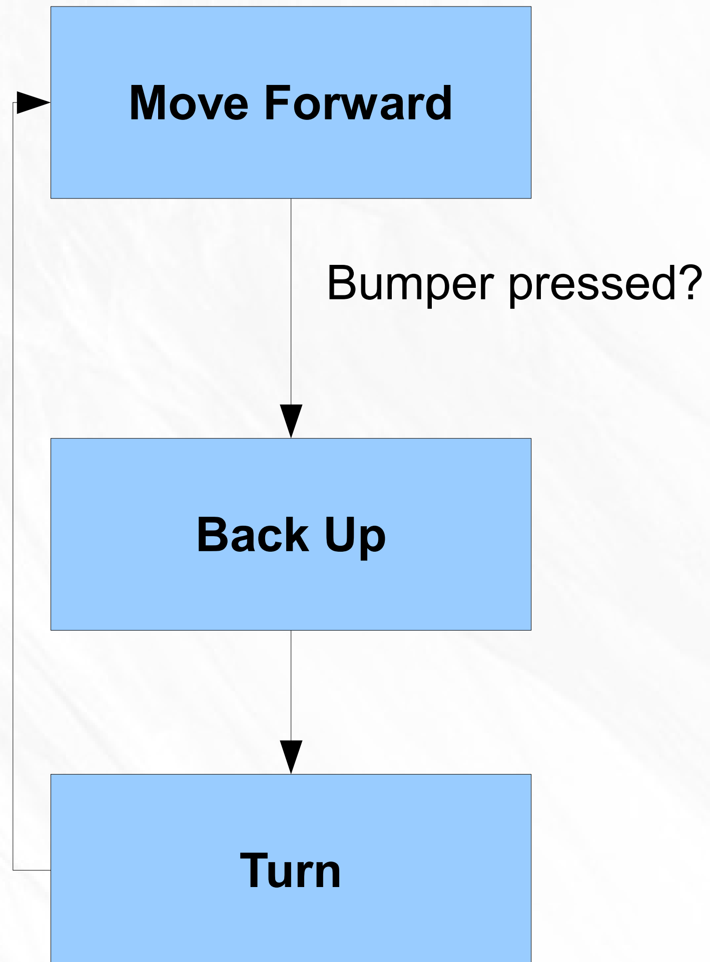
Example

- Light switch
  - Two states – on and off
  - Transition between these states is dependent upon input

Robot Example

- Roomba
- States
  - Move forward
  - Back up
  - Turn

Diagram



Behavior Programming

- A Behavior is a state or set of states that act a specific function
- Behavioral programming is built into the Lejos (Java firmware) package for Lego NXT.

Pros and Cons

- Pros
  - Cleaner code
  - Easier to handle transitions
- Cons
  - Not as intuitive as an FSM
  - Sometimes harder to control exactly what happens if state transitions are complex

Behavior API

- One interface – Behavior
- One class - Arbitrator

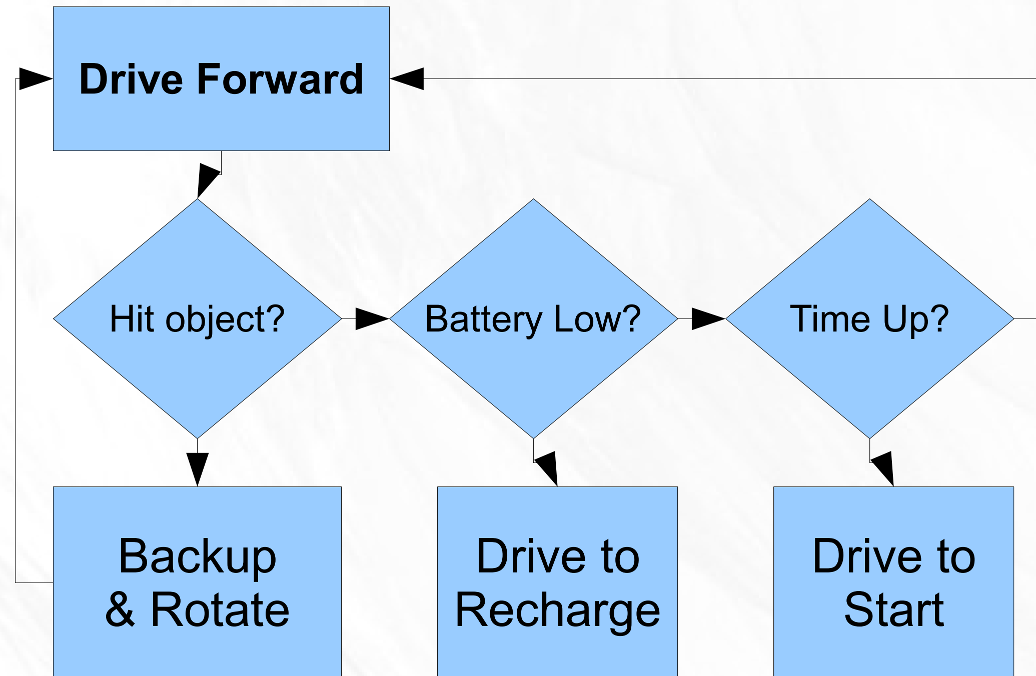
Behavior Interface

- TakeControl(). Returns a boolean on whether this behavior should become active
- Action(). Code initiates when behavior becomes active
- Suppress(). Used to terminate code and actions running from the action method

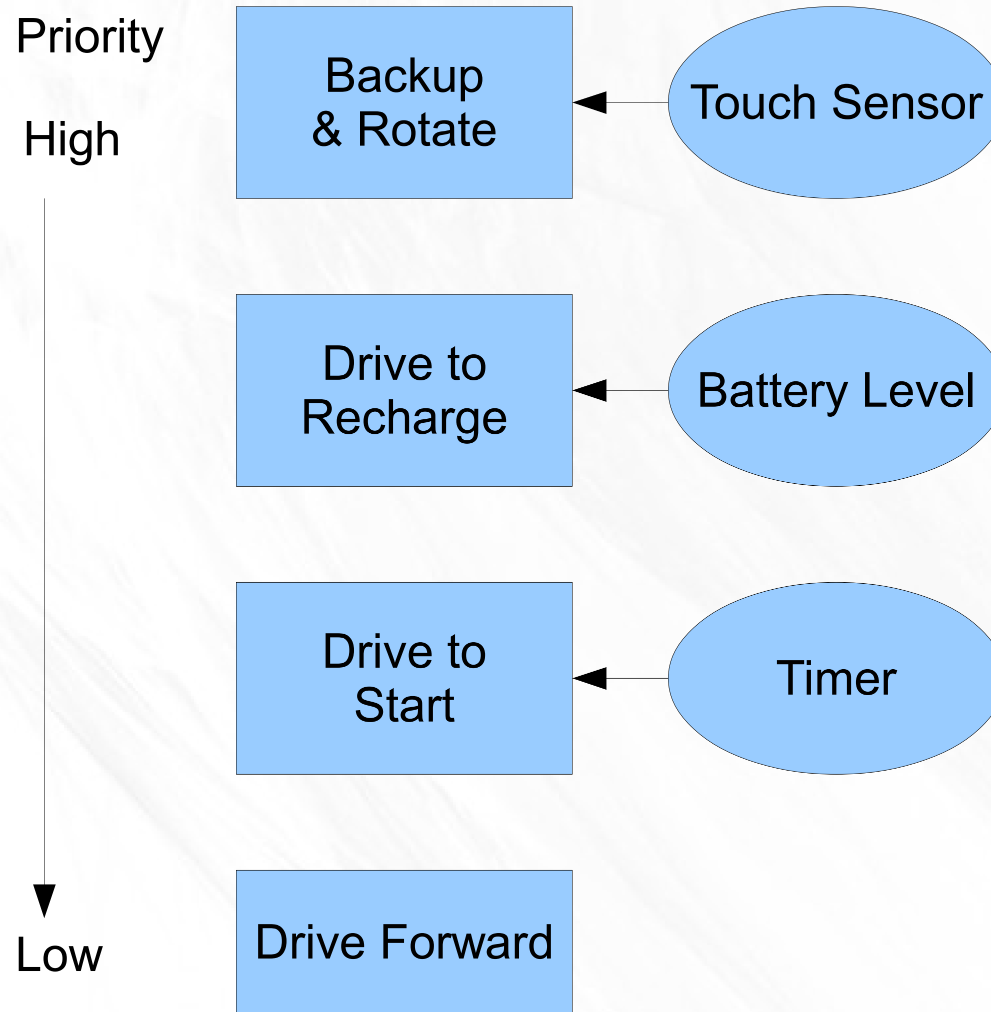
Arbitrator

- Takes a list of behaviors and regulates when the behaviors are active. The higher the index in the array, the higher the priority level.

Structured FSM Example



Behavior Method



Graphs

- Definition
- A graph G can be defined as the set of nodes or vertices N, linking with a set of edges E.
- G = {N, E}

Graph Density

- The ratio of edges and nodes indicate whether a graph is SPARSE or DENSE.
- Prefer a sparse graph when possible.
- Knowledge of whether a graph is dense or sparse is helpful when selecting the appropriate data structure to encode the graph.

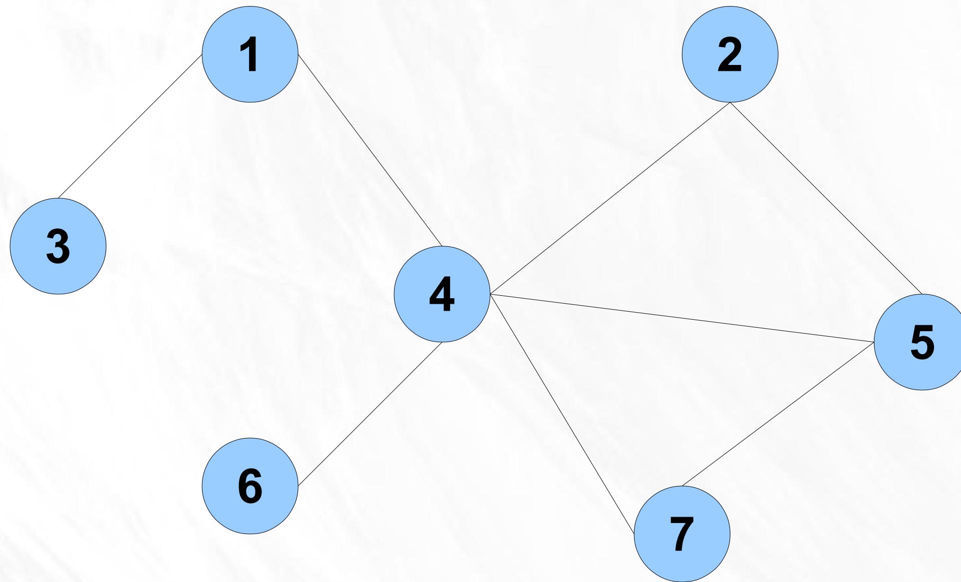
DiGraph

- Can create a graph that assumes all edges between nodes are bi-directional.
- Sometimes not the case
- Example
  - May have a door between two rooms, this would be an edge where the rooms would be the nodes
  - Sometimes a door is only one direction – typically, used to allow enemies to come into the room, but nobody can exit to the other room.

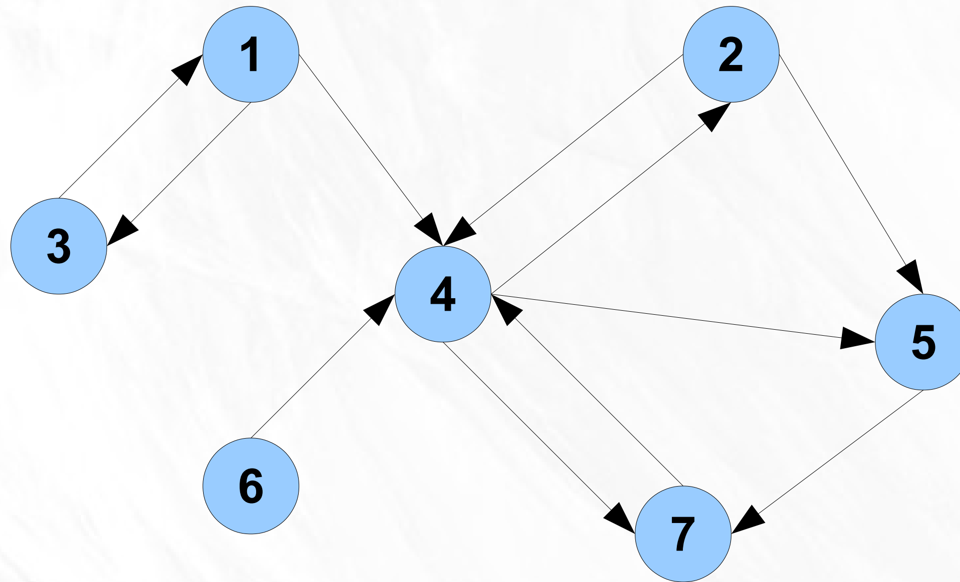
DiGraph - continued

- Sometimes may be possible to travel between two nodes in either direction, but the cost is different.

Regular Graph Example



Di-Graph Example



Navigation Graphs

- Abstraction of all locations in an environment the agents may visit and of all connections between those points
- Each node represents a key area within the environment and each edge represents the connections between these two points
- Each edge will have an associated cost
- Simplest case – distance between the nodes
- Known to mathematicians as a Euclidean graph

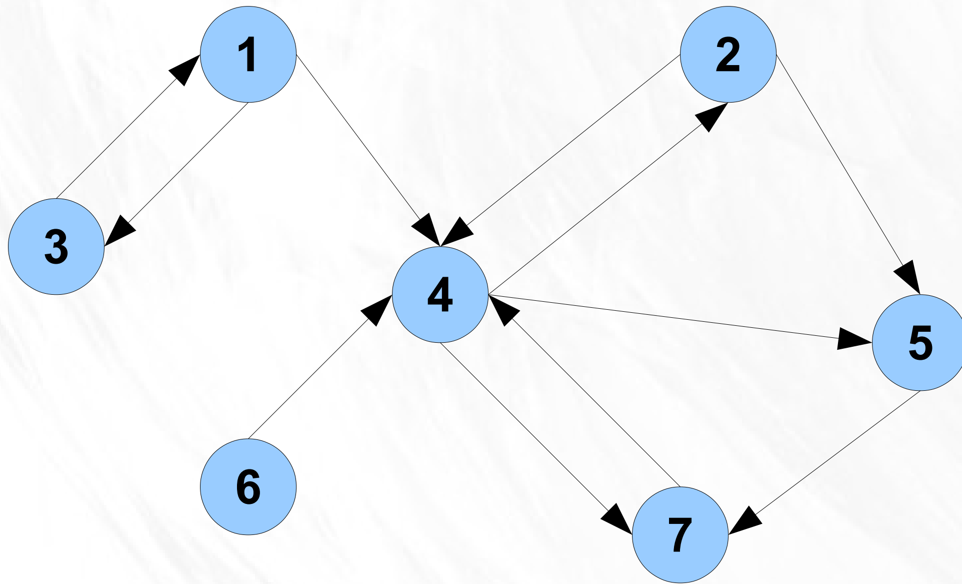
Implementation

- Adjacency matrix
- Adjacency list

Adjacency matrix

- Matrix T
  - Two dimensional matrix of booleans or floats to store a graph's connectivity information

Example



	1	2	3	4	5	6	7
1	0	0	1	1	0	0	0
2	0	0	0	1	1	0	0
3	1	0	0	0	0	0	0
4	0	1	0	0	1	0	1
5	0	0	0	0	0	0	1
6	0	0	0	1	0	0	0
7	0	0	0	1	0	0	0

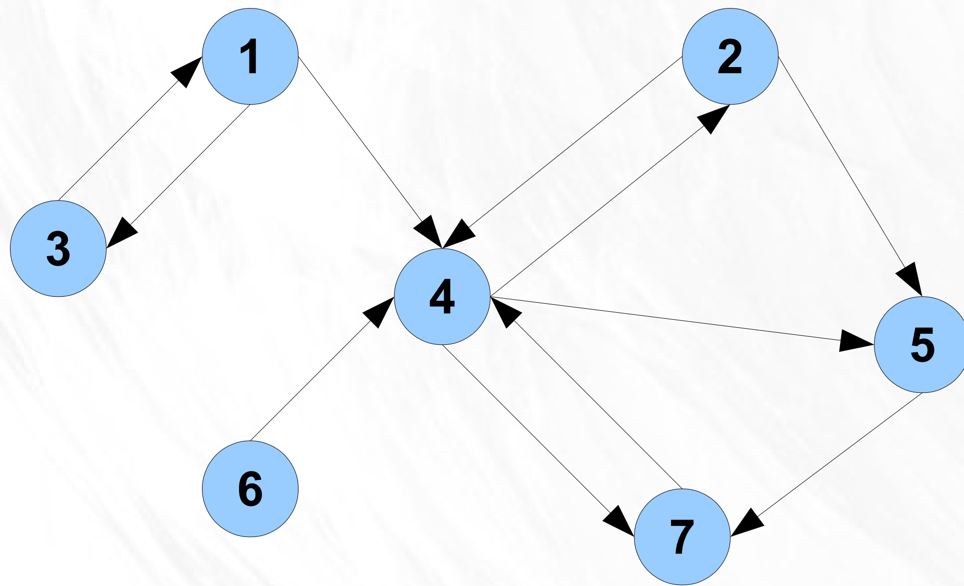
Notes

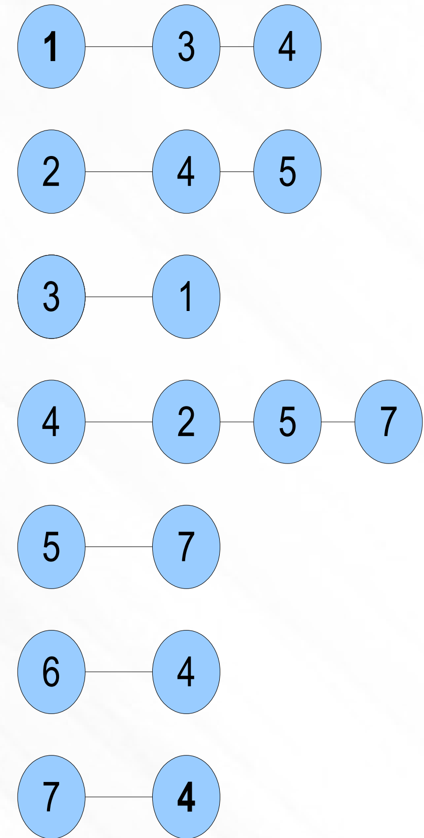
- Ineffecient in a sparse graph due to wasted memory space
- Faster to read due to direct lookup of indices in the table

Adjacency List

- Each node stores a linked list of all adjacent eges

Example





Pros & Cons

- Pros
  - Better for sparse graphs
  - More efficient use of memory space
- Cons
  - Slower to read due to iterating through lists vs direct lookup

Graph Search Algorithms

- Possible to
  - Visit every node in a graph, mapping out graph's topology
  - Find any path between two nodes.
    - Useful if want to find a node but don't care how to get there
  - Find BEST path between two nodes
    - shortest path
    - most energy efficient

Types of Graph Searches

- Uninformed
  - Depth First Search
  - Breadth First Search
- Cost-Based
  - Djikstra's Algorithm
  - A*

Depth First Search

- Searches by moving as deep into the graph as possible.
- When it hits a dead end, backtracks to a shallower node where it can continue

Breadth-First Search

- Fans out from the source node and examines each of the nodes its edges lead to before fanning out from those nodes and examining all the edges they connect and so on.
- Finds the shortest path with fewest nodes
- Cons – so systematic, it can prove unwieldy to use on anything other than small search spaces

Cost-Based Graph Searches

- For many applications, related graph will have a COST associated with traversing an edge

Edge relaxation

- As an algorithm proceeds it gathers information about the best path found so far from the source node to the target
- This information is updated as new edges are examined
- If a newly examined edge infers that the path to a node may be made shorter by using it in place of an existing best path, that edge is added and the path is updated accordingly

Shortest Path Trees

- Given a graph G, and a source node, the shortest path tree is the sub-tree of G that represents the shortest path from any node on the SPT & the source node
- Used in the following algorithms, which find the shortest path tree in weighted graphs by “growing” a shortest path tree outward from the source node

Djikstra's Algorithm

- Builds a SPT one node at a time
- Adding source node to SPT and then by adding the edge that gives the shortest path from the source to a node NOT already on the SPT.
- Uses Indexed Priority queue

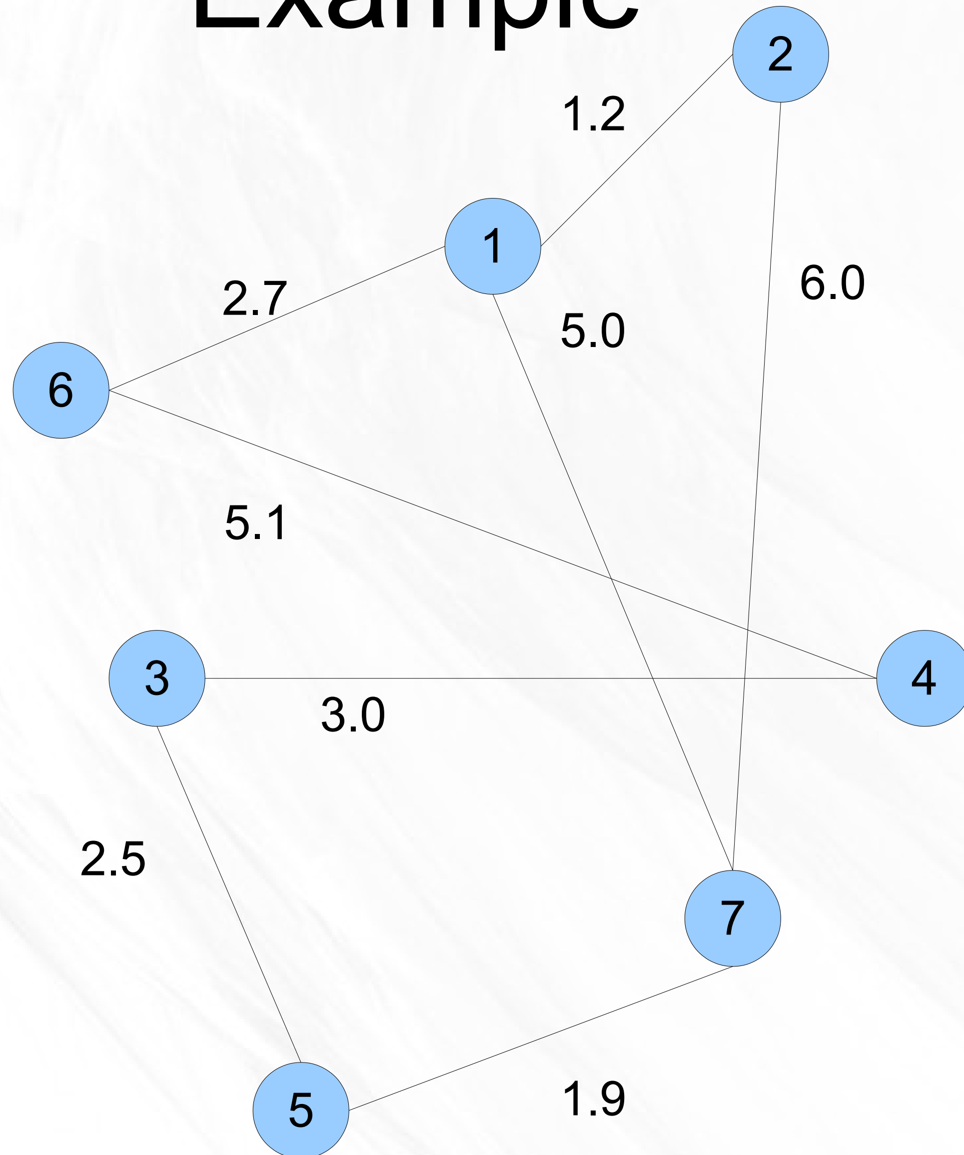
A*

- Djikstra's algorithm can be improved significantly by taking into account an ESTIMATE of the cost from each node under consideration when putting nodes on a search frontier.
- This estimate is called a HEURISTIC
- If the heuristic used by the algorithm gives the LOWER BOUND on the actual cost (underestimates), the A* is guaranteed to give optimal paths
- For navigation graphs, most straight forward heuristic is the straight-line distance between nodes
- This distance is called Euclidean distance

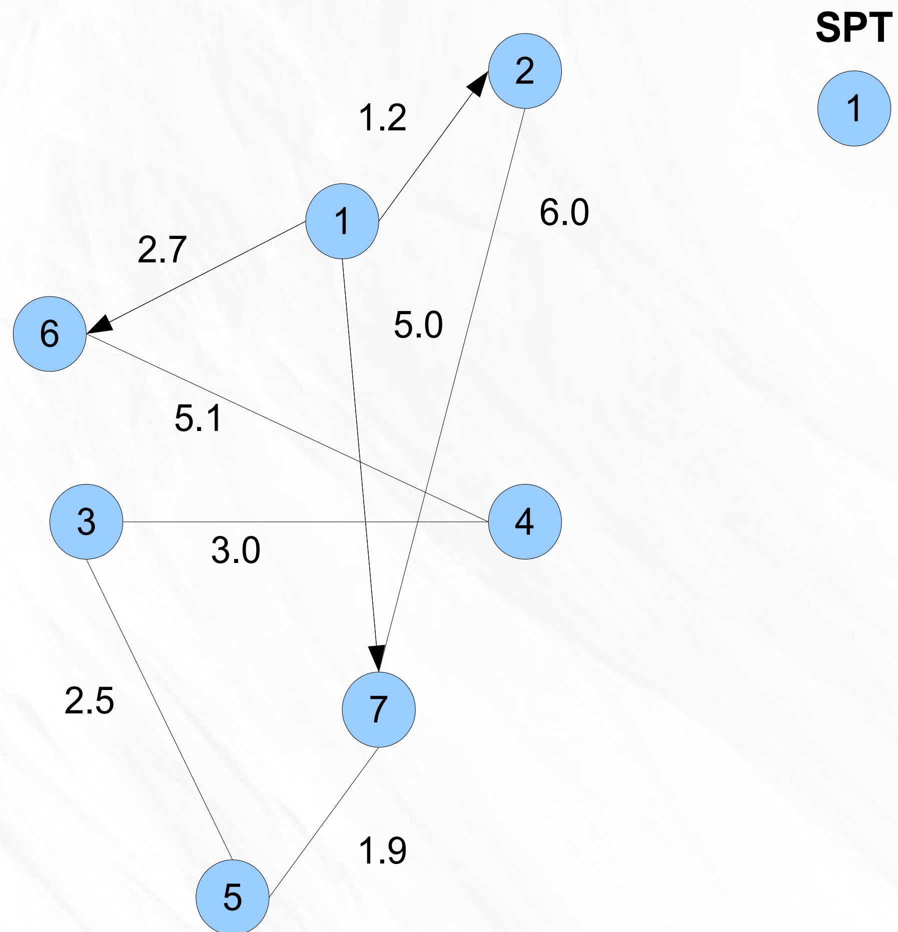
Calculating A*

- Adjusted cost of nodes on the search frontier when positioned on the priority queue.
- F = G + H
  - G is the cumulative cost to reach a node
  - H is the heuristic estimate of distance to the target
- Results in fewer edges that need to be examined
- Con of A* - finding the correct heuristic is extremely important.

Example



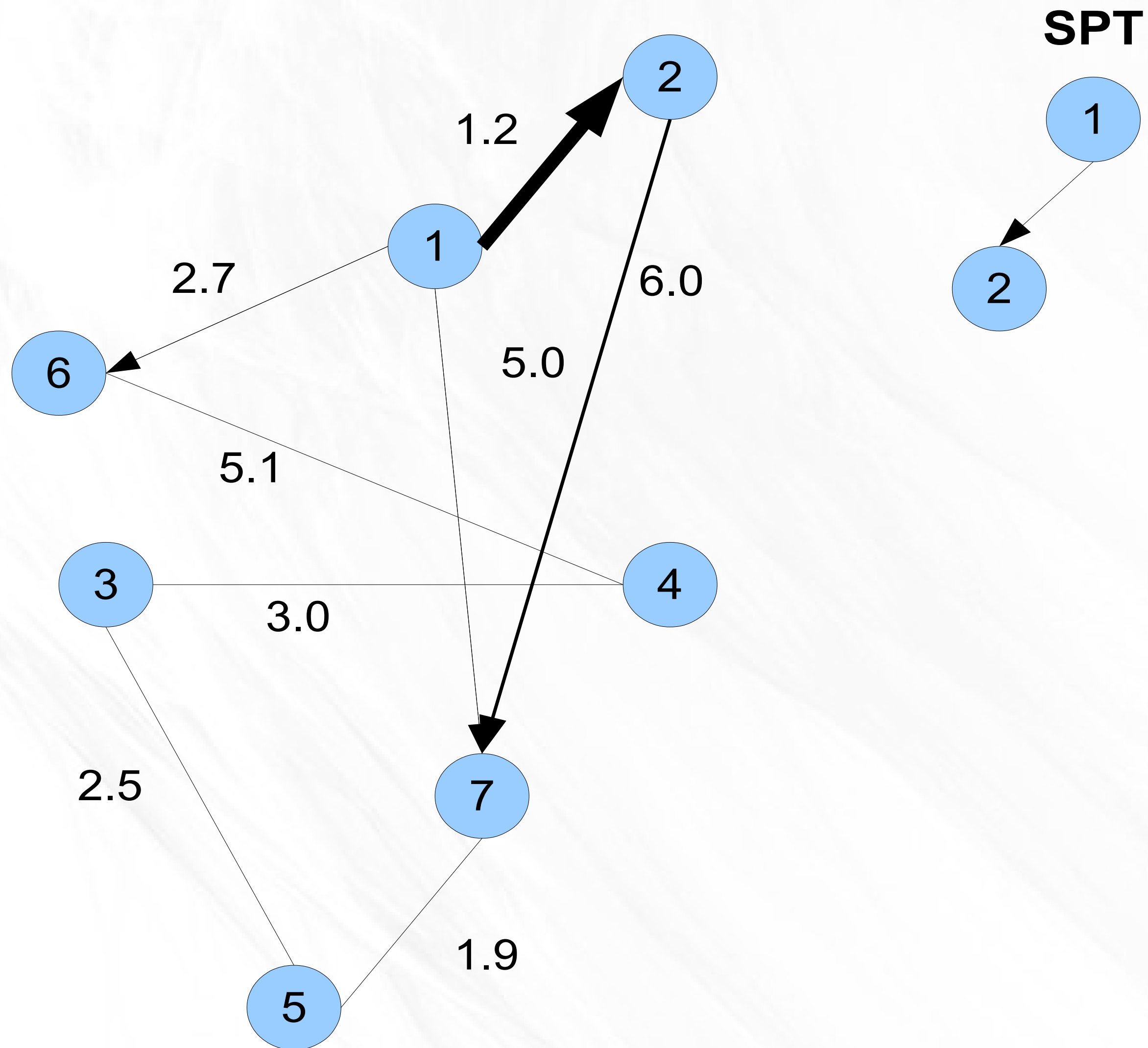
Example – cont



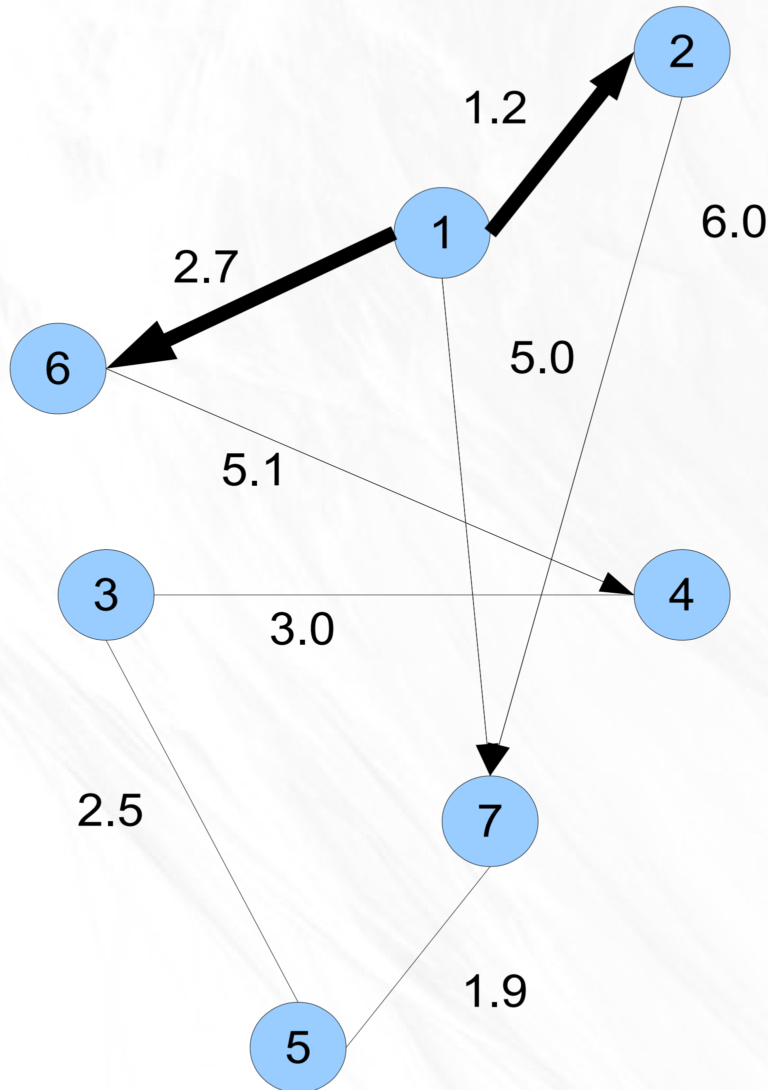
SPT

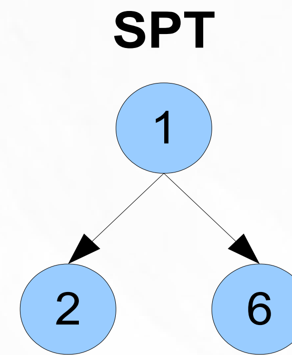
1

Example – cont

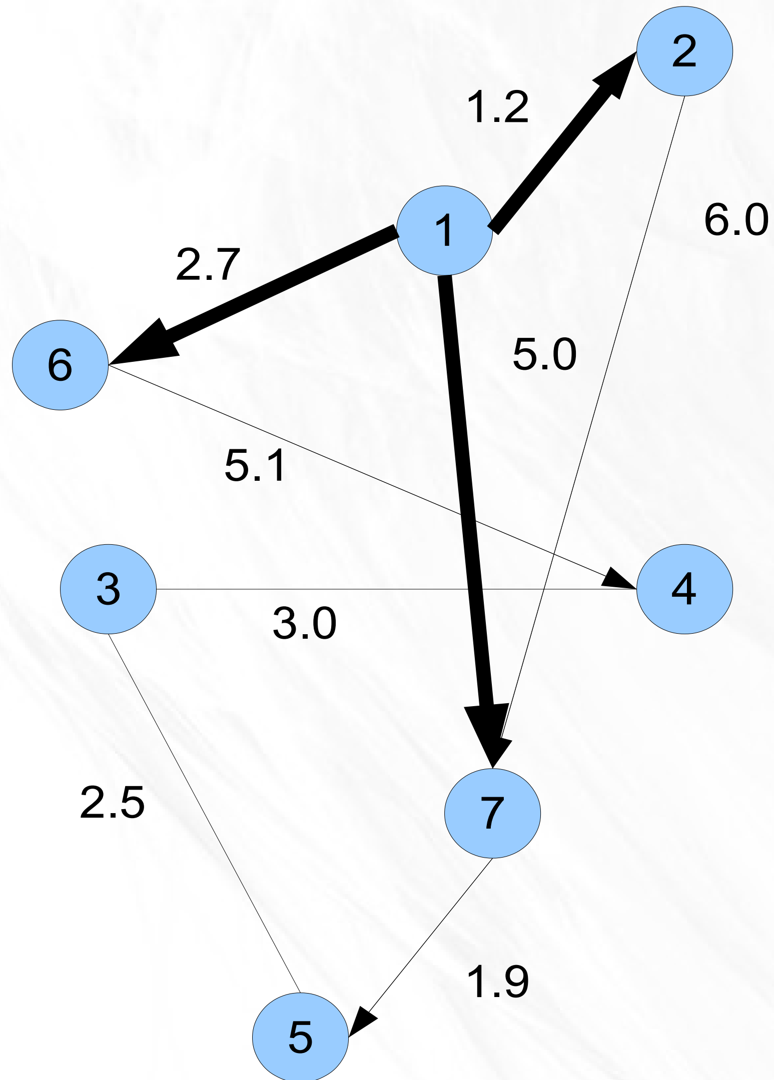


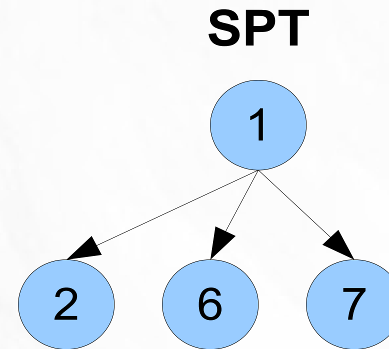
Example – cont



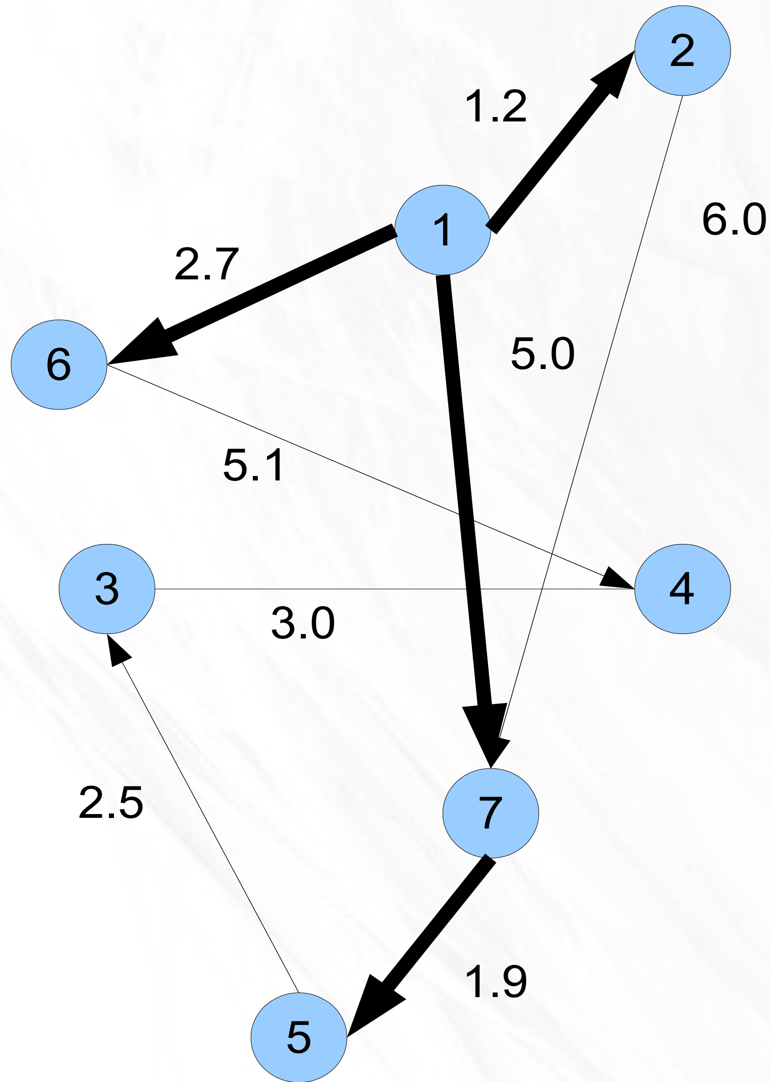


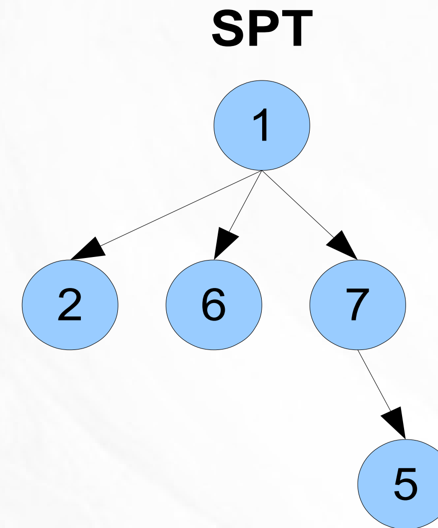
Example – cont





Example – cont





Example – cont