What We Want (& Will Get)

1. *Consistent Performance:* If we run "A^*_h" twice in a row, we want to get the same nodes both times.
2. *Monotonic Improvement:* Given an "improved" heuristic h_1, we want "A^*_{h1}" to be at lest as efficient as "A^*_h".
3. *Heuristic Equivalence*: Given "$A^{*'}_h$", we want them both to expand the same nodes.
4. *Optimal Efficiency:* We want it to be at least as efficient as any other optimal search algorithms that h dominates.

How can we do this?

- We could “modify” A*.
- We could place more restrictions on the heuristics being used.
- We will look at both ways.

Modifying A*

- What type of modification should we be looking at?
- At least part of the problem seems to be that the handling of critical ties is underspecified.
- We need to refine our definition of A* to describe how ties are handled.

Consistent Performance

- How is it possible to get different nodes when running the same algorithm twice?
- Make the selection of nodes from the open list determined solely by either the structure of the search tree or by its traversal.
- Then will get consistent performance.

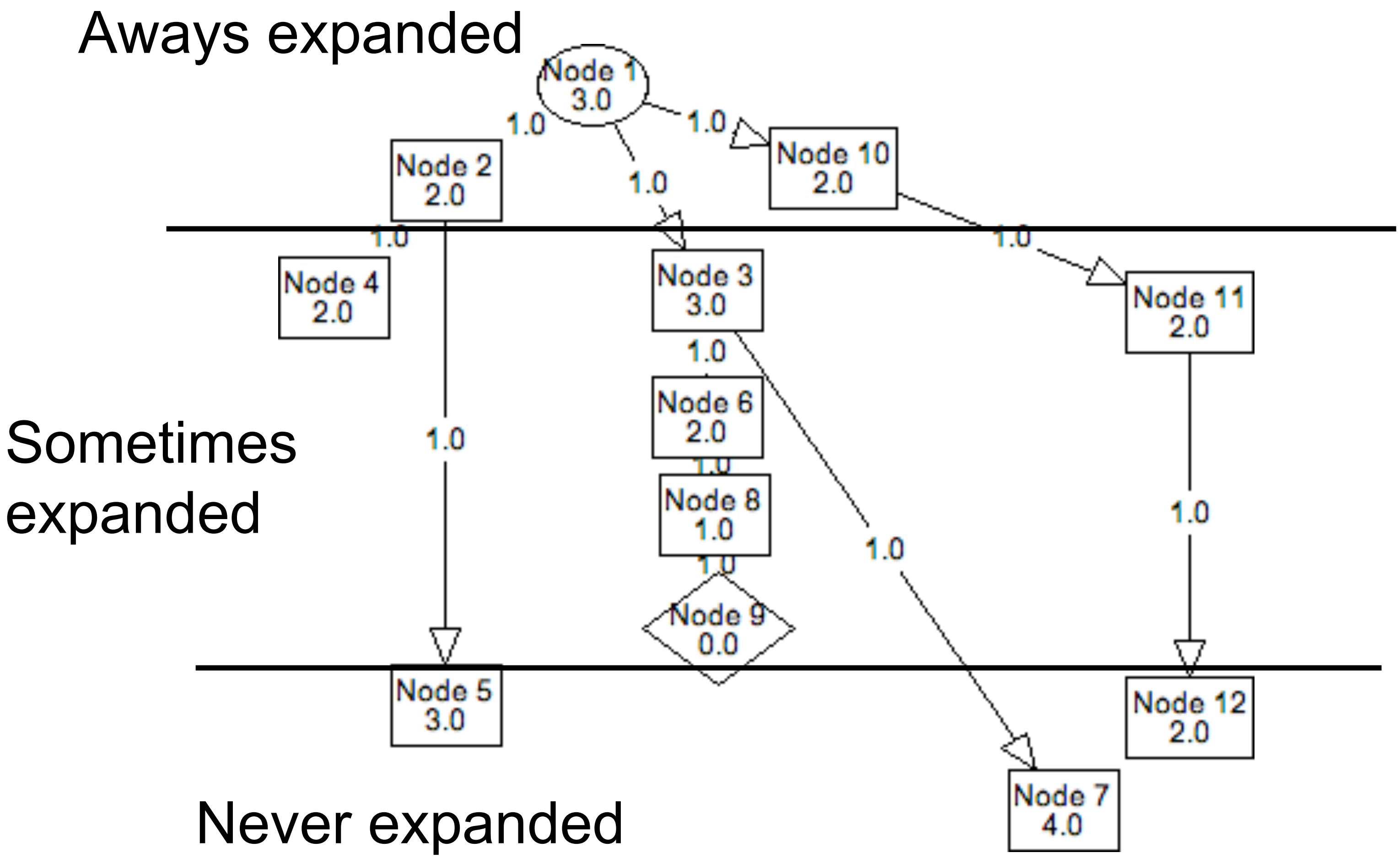
Monotonic Improvements

- Why don’t we currently have monotonic improvements?
- To understand this we have to look at how current open list access mechanisms lead to this problem.

Common Open List Access Mechanisms

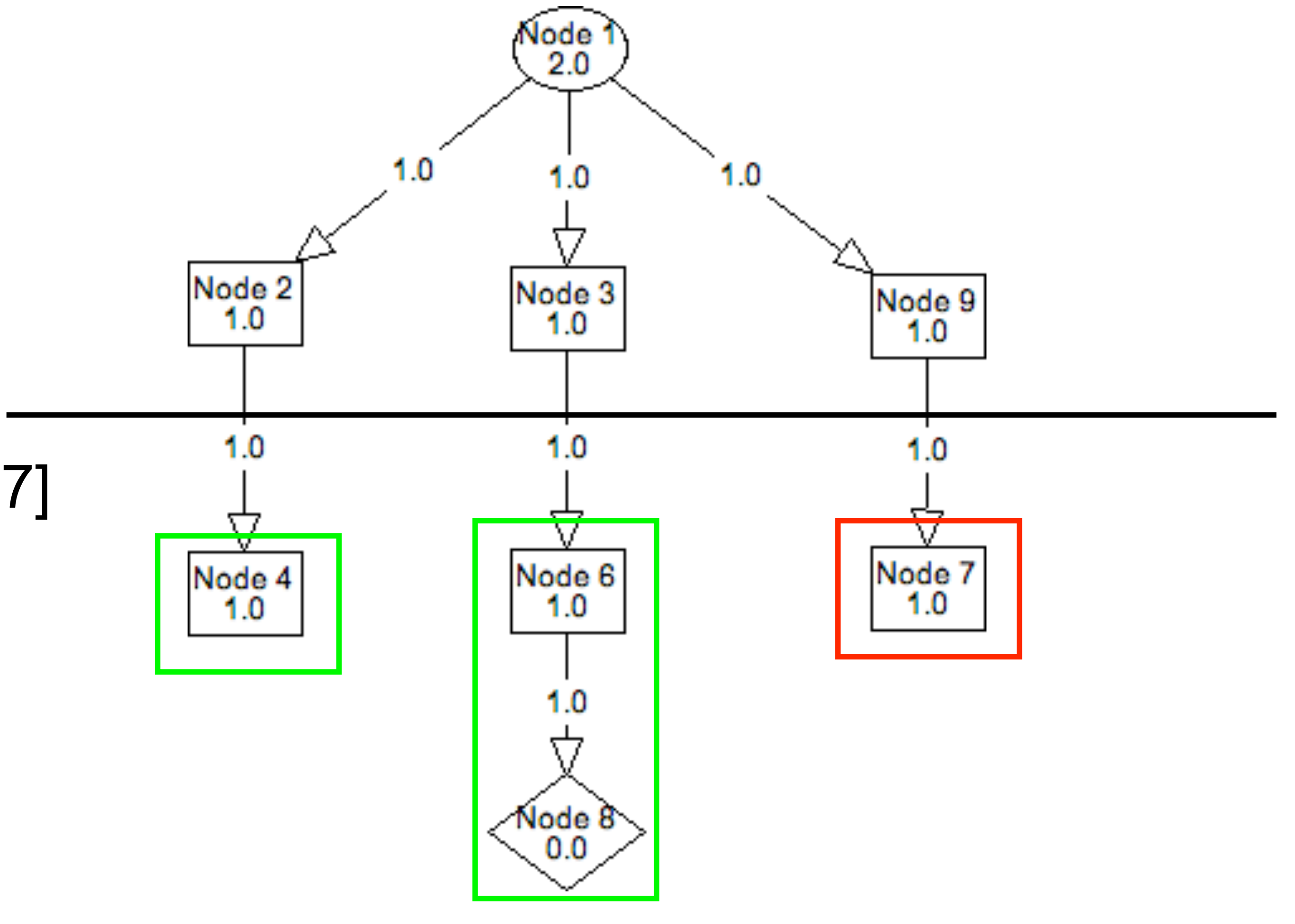
- Stack
- Queue
- Both of these leads to non-monotonic improvements.
- We will now look at why.

Search Tree Expansion

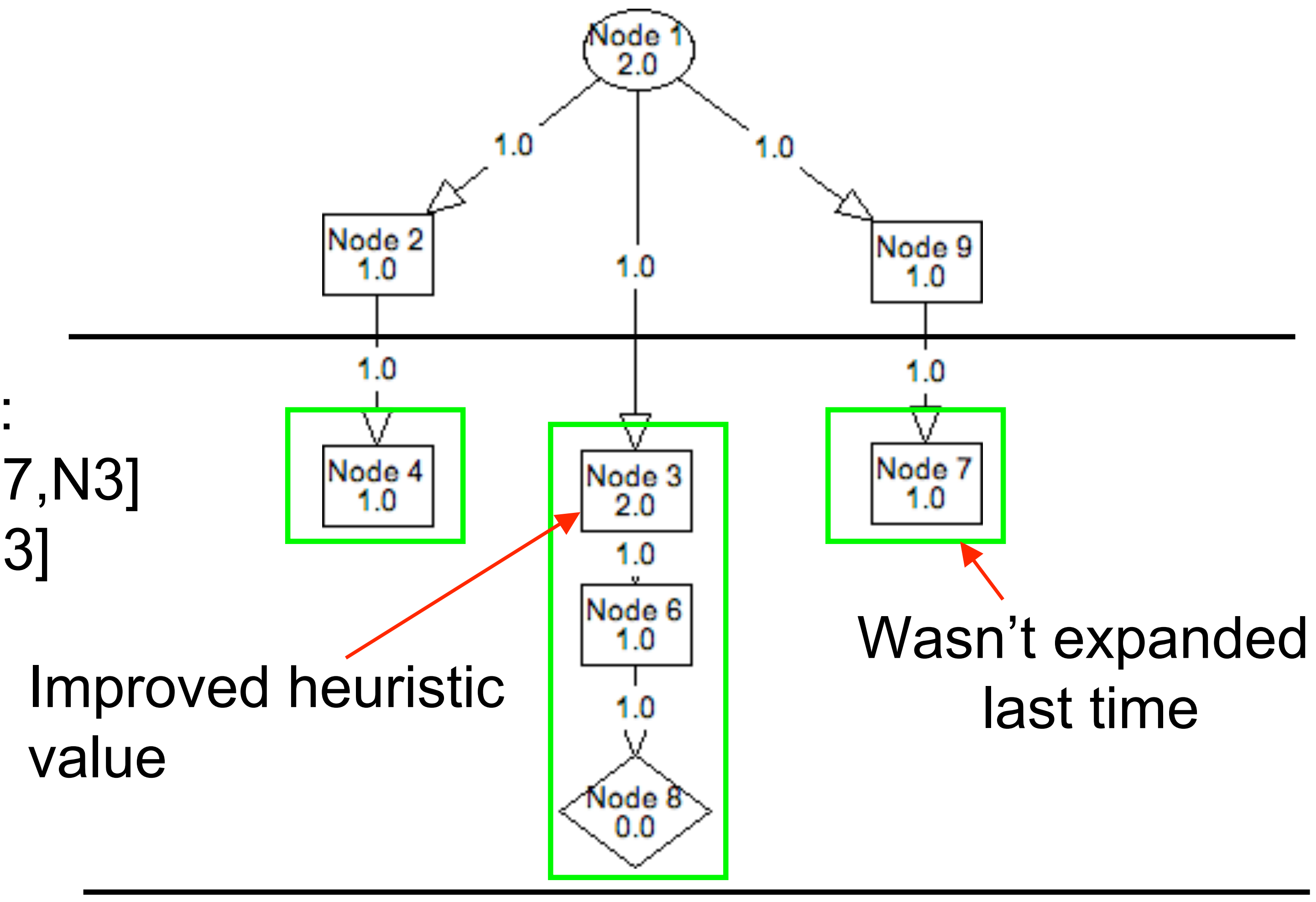


Stack Access Mechanism

Stack:
[N4,N6,N7]
[N6,N7]
[N8,N7]



Stack Access Mechanism



Stack:
[N4,N7,N3]
[N7,N3]
[N3]
[N6]
[N8]

Access Mechanism Problem

- Saw example of using a stack access mechanism, where an improved heuristic led to a loss of efficiency.
- We can do the same sort of thing for queue access mechanisms.
- Why do these access mechanisms have this type of problem?