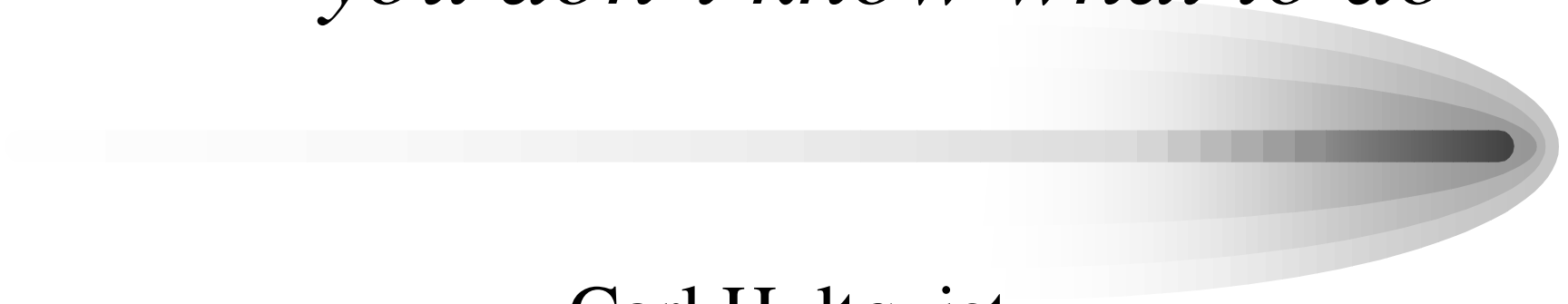
Heuristics, and "what to do if you don't know what to do"



Carl Hultquist

What is a heuristic?

- “Relating to or using a problem-solving technique in which the most appropriate solution of several found by alternative methods is selected at successive stages of a program for use in the next step of the program”
- “A common-sense rule (or set of rules) intended to increase the probability of solving some problem.

Enough definitions! What are some examples of heuristics or their uses?

- Greedy algorithm
  - most well known, and well used
- Heuristic searches
  - classic examples are:
    - A*
    - IDA* (**I**terative **D**eepening A*)
    - RBFS (Recursive Best-First Search)

The Greedy Algorithm

- Simply pick the best thing to do. Do it, then repeat until you reach your goal.
- More formally:
  - from state **S**, generate the set of all the *successor* states, **N**
  - for each state in **N**, determine a score for the state (based on how close you think it is to the goal state)
  - pick the state in **N** that has the best score

Heuristic searches

- Used in searching through graphs, usually where you are given a starting state and need to reach a goal state
- All have one thing in common: a heuristic *function*
- A heuristic function has the form:

$$f^(n) = g^(n) + h^(n)$$

where n represents some state, g^(n) is the length of the path to state n, and h^(n) is the estimated distance of state n from the goal state. h^(n) is the heuristic part: this is usually an estimate.

Heuristic searches: an example

2	8	3
1	6	4
7		5

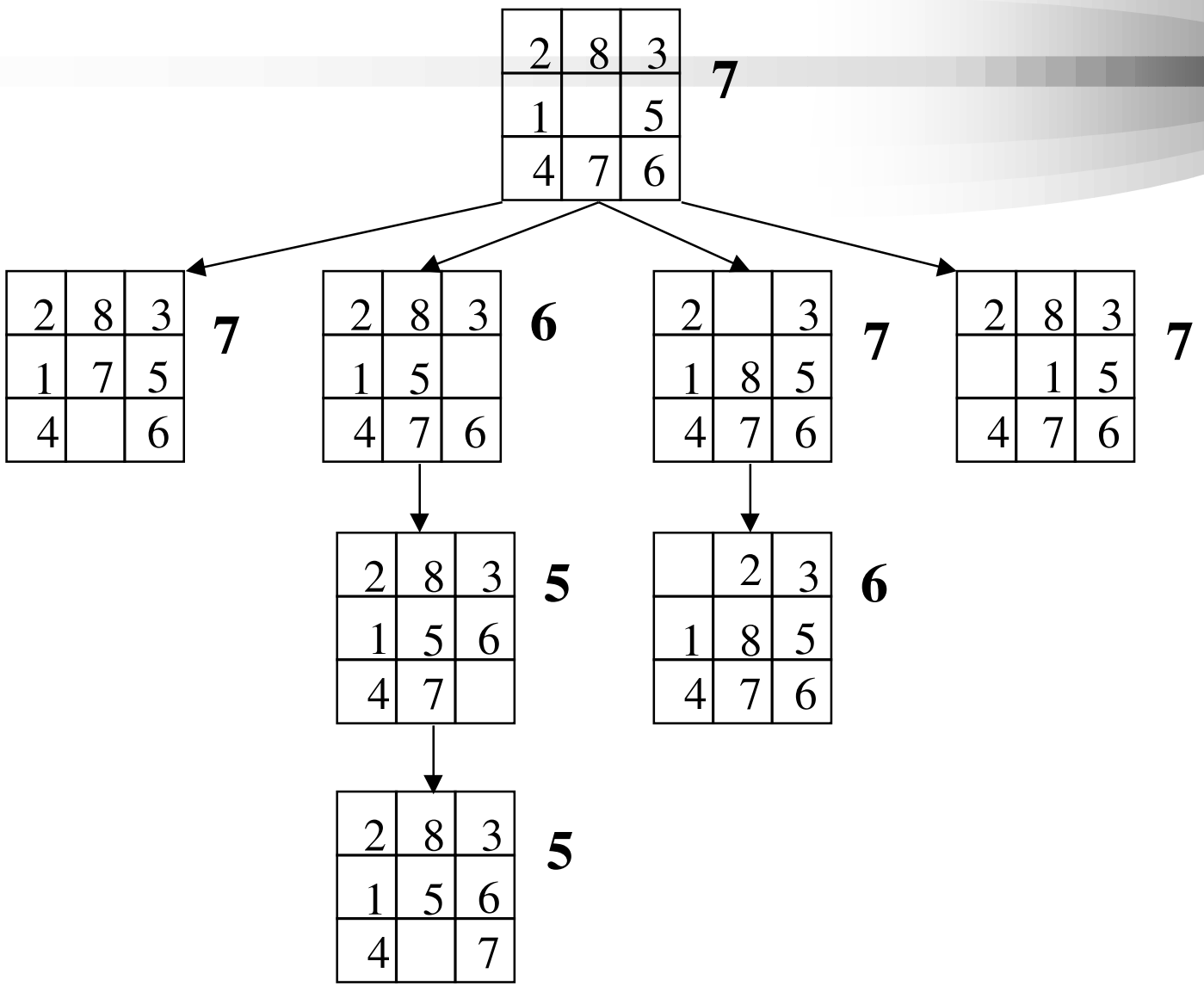
- How can we design a heuristic for the sliding tile puzzle?

Answer: using what is known as the *Manhattan distance*. For each tile, compute how many moves it would take to move that tile to its goal position if there were no other tiles on the board. Add up these distances for all of the tiles: this gives a rough idea of how far this state is from the goal state.

So in this example, the Manhattan distance for the state shown is 1 + 1 + 0 + 2 + 2 + 1 + 2 = 9

Another well-used heuristic is to simply count the number of tiles that are not in their correct positions. In this case, that is 6 tiles.

So what's the g'(n) useful for?



The A Algorithm*

- Keep a list of *active* states. Start by putting the initial state into this list, then repeat the following:
  - Pick the active state with the lowest f^ value, call it state **S**
  - If **S** is the goal state, then we've found the best path. Exit!
  - Remove **S** from the list
  - Generate all the successors of this state. If the path from the initial state through **S** to any of these states is better than one we're found before, then update the successor state to remember that the best path to it was from **S**. Add those successor states that we have not processed before to the active list.

Comments on the A Algorithm*

- If a path to the goal exists, then A* will always find the shortest path to the goal
- Because A* closely resembles a breadth-first type of search, it consumes lots of memory

*An alternative: IDA**

- IDA* = Iterative Deepening A*
- Performs a series of depth-first searches, each with a certain cost cut-off. When the f^ value for a node exceeds this cut-off, then the search must backtrack. If the goal node was not found during the search, then more depth-first searches are conducted with higher cut-offs until the goal is found.

RBFS (Recursive Best-First Search)

- As the name implies, a form of recursive searching
- As each node is processed, change the f^ values of each node in the tree to be the minimum of the f^ values of its children
- Then pick the node with the smallest f^ value and follow that path
- Sort-of like a depth-first version of A*, in that we always pick the node with the smallest f^ value in the whole tree. It's different in that we *propagate* small f^ values back up the tree

Extra info on heuristics

- Often used in artificial intelligence, since humans often behave based on heuristics rather than planning the absolute best way of doing something
- Good book is “Artificial Intelligence: A New Synthesis” by Nils J. Nilsson

In olympiads: what to do if you don't know what to do!

- Don't panic!
- Use heuristics
- Use a naïve algorithm
- Use a naïve algorithm with a timeout

Using heuristics

- Key checkpoints before using heuristics:
  - does the problem allow me to get partial marks?
  - can I measure how close a solution is to the goal?
- Other things that will help:
  - can I spot a quick way of finding any solution (possibly sub-optimal)?
  - how would a human solve this problem? Can I make the computer do something similar?

Using naïve algorithms

- Better than heuristics if you can't get partial marks, although also useful if you can get partial marks.
  - Without timeout: best for no partial marks. Means that your program will probably run out of time for later test-cases, but at least you'll get the small ones
  - With timeout: best for partial marks. Try and find the best solution possible, then when you're out of time write out the best answer you've found and exit.

An example: the sliding tile puzzle

- **Problem**: given a sliding-tile puzzle of size **N**, find the shortest sequence of moves to slide the tiles into the correct places.
- Could do a breadth-first search for a 3x3 puzzle, but anything larger starts consuming too much memory.
- In general, the sliding tile puzzle is considered *intractable*, and to this day is used as a good test for artificial intelligence algorithms. If you were given this in an olympiad, where would you start?

Ideas for solving the sliding-tile puzzle

- Firstly, let's evaluate if heuristics would be a good choice to use here:
  - since researchers can't find a good solution to this problem, we can guess that neither can the contest organisers! This means that there should be some kind of partial marks scheme, depending on how good your solution is compared to that of the organisers.
  - it's very easy to determine how close a solution is to the goal, by either using the Manhattan distance or the number of tiles that are out of place
- This means that heuristics should work quite well.

Ideas for solving the sliding-tile puzzle (2)

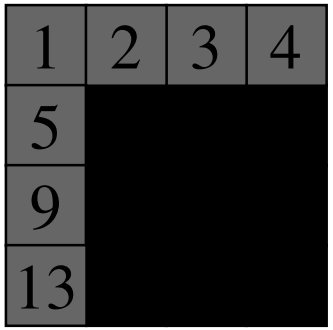
- Next step: what kinds of things could we try?
  - A heuristic search. In this case, you'd probably have to use a depth-first type of search - either IDA* or RBFS - since for N greater than 3 the A* algorithm would take up too much memory. But even the IDA* or RBFS are likely to be a bit slow with larger N, so if you used these you might need to use a timeout as well.
  - For small N (like 3 and maybe 4), you could do a breadth-first search. This will guarantee that you get these cases 100% right. The down-side: this won't work for bigger cases of N.

Ideas for solving the sliding-tile puzzle (3)

- We still need to find something that will work for all N: let's work out how a human might do the puzzle.
  - Most humans probably solve the puzzle by moving the tiles to their correct spots one at a time.
  - Special care needs to be taken for the corner pieces.

Ideas for solving the sliding-tile puzzle (4)

- Divide and conquer!
  - If you think about it, a 4x4 puzzle is the same as a 3x3 puzzle with an extra layer wrapped around it. In general, an NxN puzzle is the same as a (N-1)x(N-1) puzzle with an extra layer wrapped around it.
  - So to solve an NxN puzzle, all we need to do is move the pieces of the extra layer into their right places and then solve the remaining (N-1)x(N-1) puzzle.



Final strategy for sliding-tile puzzle

- Process 1 layer at a time. For each layer:
  - move one tile at a time into position
  - process tiles in the layer in order (so start at one end of the layer, and work toward the other)
- Then, if you have time, do the following extras:
  - Once you reach a 3x3, do a breadth-first search to finish off the last part of the puzzle in the smallest number of moves
  - Start generating random paths to try and find a shorter path than the one you constructed from the previous steps. Put in a timeout to make your program output the best path it found when it runs out of time. Or use some kind of depth-first search like IDA* or RBFS.

Extra ideas

- Another idea is to consider each tile as an *agent* that has its own goal (to get to the correct position). Each tile can then either force other tiles to move out of its way, or another tile might attack it and force it to move out of the way!
- This kind of “intelligent” approach can lead to less “logical” solutions, but in so doing can avoid things like loops and can find new ways of solving a problem efficiently.

Readings on the sliding-tile puzzle

- For those interested:
  - "A Distributed Approach to N-Puzzle Solving", by Alexis Drogoul and Christophe Dubreuil
  - "Kevin's Math Page: Analysis of the 15/16 puzzle". http://kevingong.com/Math/SixteenPuzzle.html