# CPSC 322
## Introduction to Artificial Intelligence

October 25, 2004

# Things...


Assignment 3 was posted Friday night

Midterm exam 2 is one week from today

Bring me your first midterms before the second midterm if you want to be retested on problem 3

Jessica Hodgins talk, Thursday, October 28, 1-2pm, MacLeod 214

# Game search

(also known as adversarial search) has these components:

move (or board) generator

static board evaluation function (this is the heuristic part - it doesn't generate moves or look ahead - it's static)

minimax algorithm to alternately propagate minima and maxima upward from "bottom"

# Minimax algorithm

Start with the following:

a) there are two players, MAX and MIN

b) it's MAX's turn to move

c) MAX has a static board evaluation function that returns bigger values if a board is favorable to MAX

d) the evaluation function gets better as the game gets closer to a goal state (else why bother to generate the game space?)

e) MAX believes that MIN's evaluation function is no better than MAX's (if that's not true, then MAX should at least avoid betting money on this game)

# Minimax algorithm

1. Generate the game tree to as many levels (plies) that time and space constraints allow. The top level is called MAX (as in it's now MAX's turn to move), the next level is called MIN, the next level is MAX, and so on.

2. Apply the evaluation function to all the terminal (leaf) states/boards to get "goodness" values

3. Use those terminal board values to determine the values to be assigned to the immediate parents:
   a) if the parent is at a MIN level, then the value is the minimum of the values of its children
   b) if the parent is at a MAX level, then the value is the maximum of the values of its children

4. Keep propagating values upward as in step 3

5. When the values reach the top of the game tree, MAX chooses the move indicated by the highest value

# 1997



**Deep Blue vital statistics:**
- 200,000,000 moves per second
- 480 custom chess-playing chips

**Garry Kasparov vital statistics:**
- 3 moves per second
- meat

# Puzzles, games, and AI

Puzzles and two-player board games have long served as a laboratory for experiments with heuristic search

Solving puzzles and playing games suggests intelligence on the part of a human player, but humans don't do this stuff the same way that search procedures do

Electronic Arts probably won't be hiring you solely on your mastery of minimax search (but it couldn't hurt)

# Puzzles, games, and AI

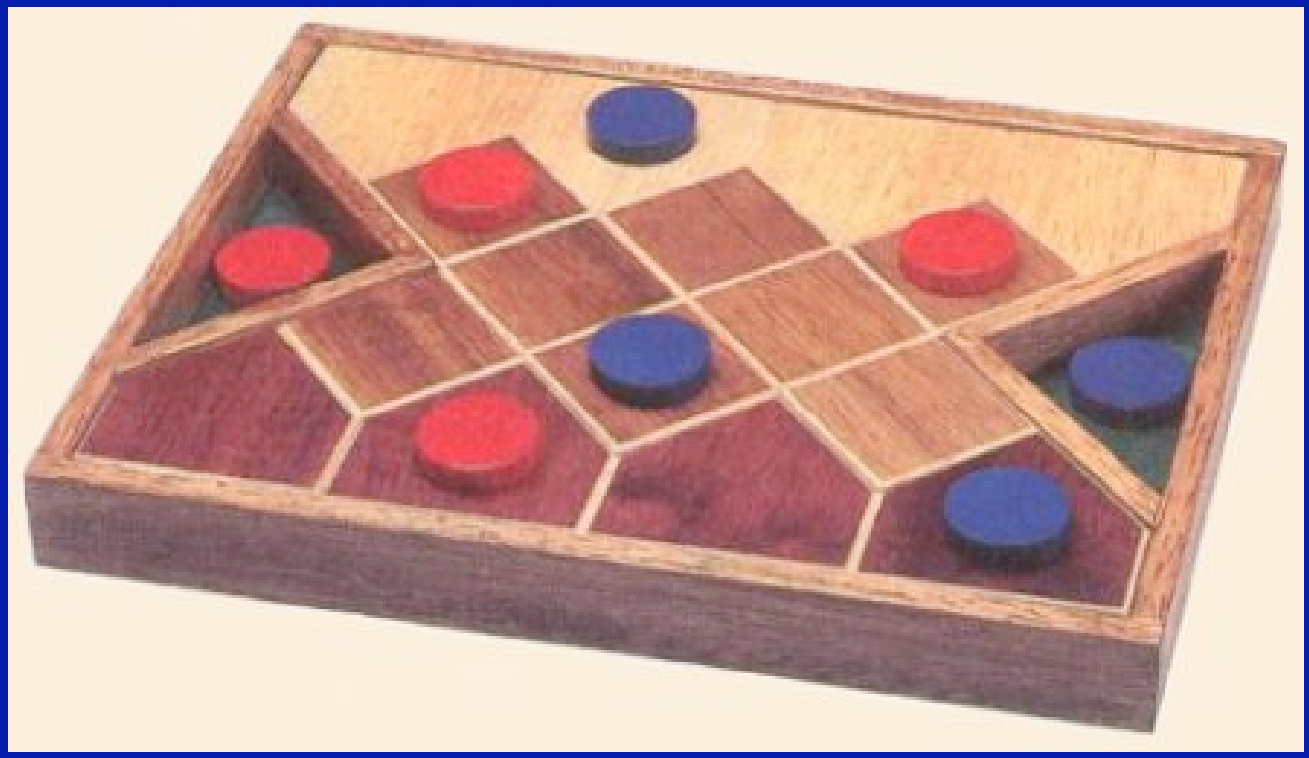
Still, heuristic search is a simple but useful tool, and the minimax game playing approach is clearly very powerful

You probably don't fully understand these search techniques until you've had to implement them...especially game search

Let's look at just one more game...

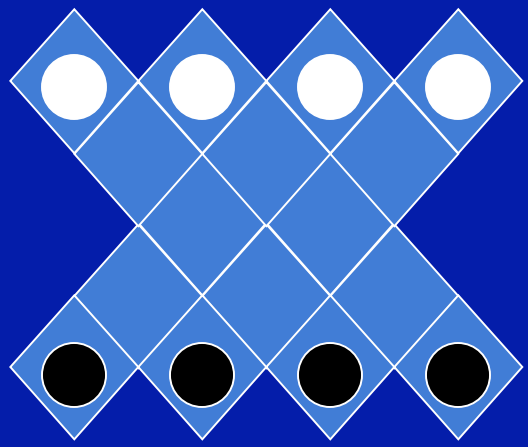
# Oska

- two players
- four pieces each



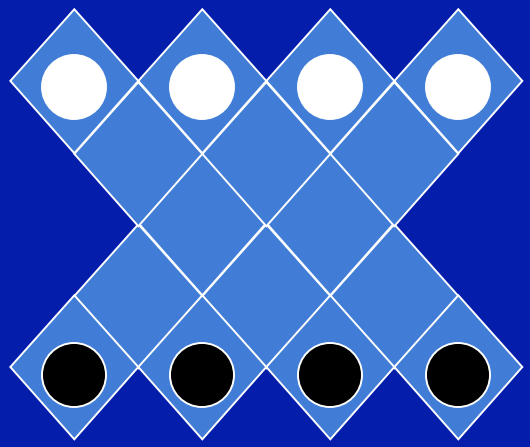
# Oska

- two players
- four pieces each



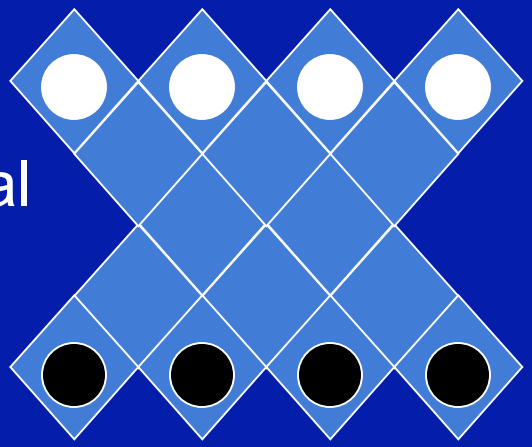
# Oska

- two players
- four pieces each
- piece movement:



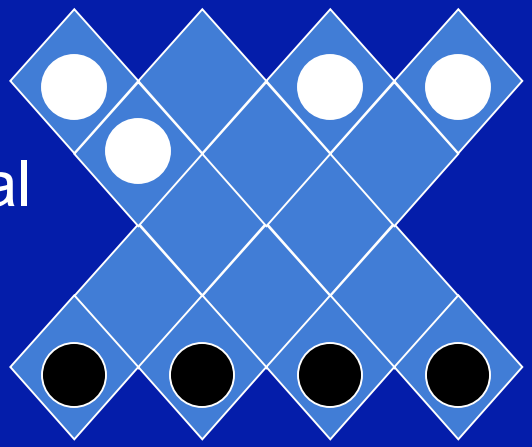
# Oska

- two players
- four pieces each
- piece movement:
  - piece may be moved one space forward on the diagonal



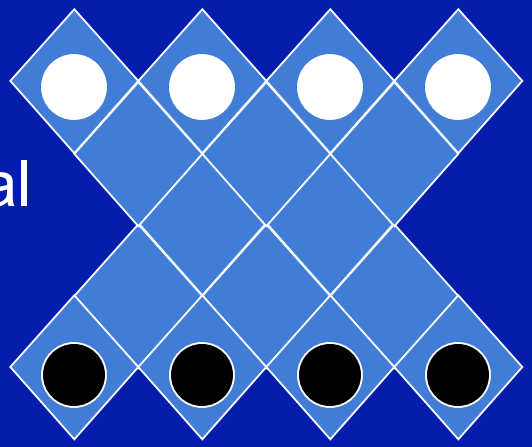
# Oska

- two players
- four pieces each
- piece movement:
  - piece may be moved one space forward on the diagonal



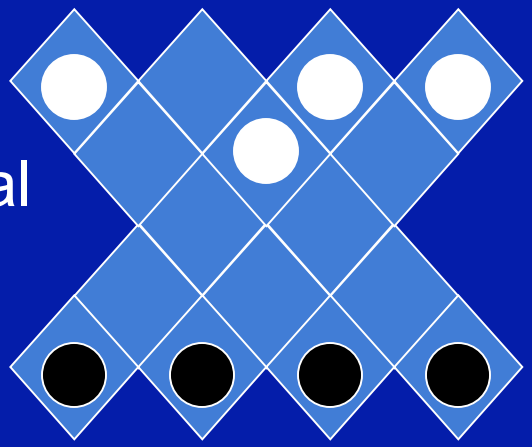
# Oska

- two players
- four pieces each
- piece movement:
  - piece may be moved one space forward on the diagonal



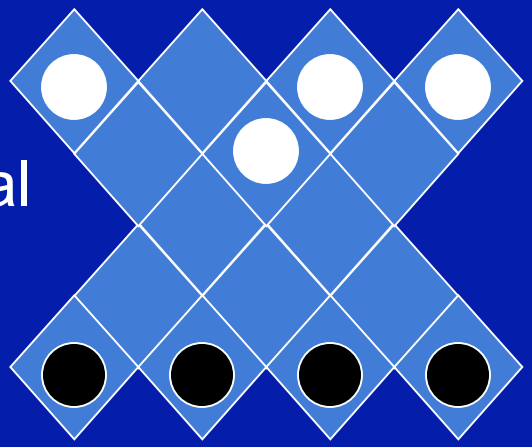
# Oska

- two players
- four pieces each
- piece movement:
  - piece may be moved one space forward on the diagonal



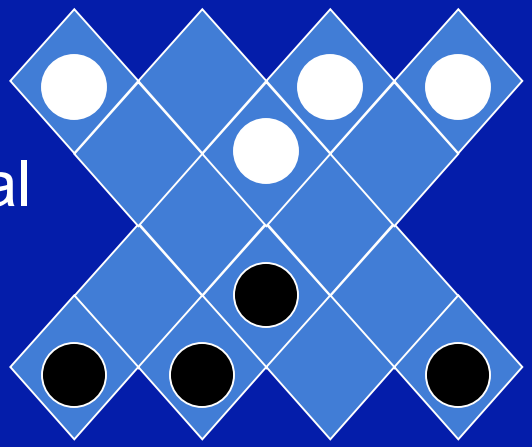
# Oska

- two players
- four pieces each
- piece movement:
  - piece may be moved one space forward on the diagonal
  - piece may jump forward on the diagonal over an opponent's piece to an empty space, thus capturing the opponent's piece



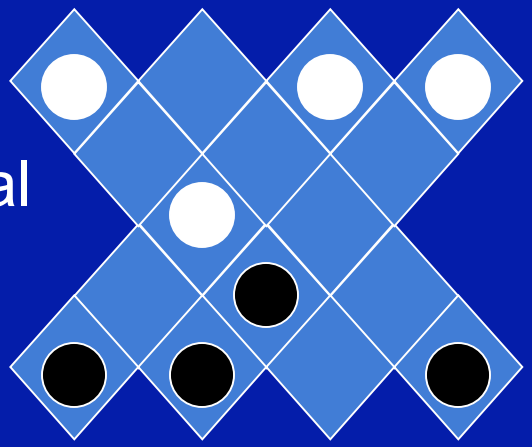
# Oska

- two players
- four pieces each
- piece movement:
  - piece may be moved one space forward on the diagonal
  - piece may jump forward on the diagonal over an opponent's piece to an empty space, thus capturing the opponent's piece



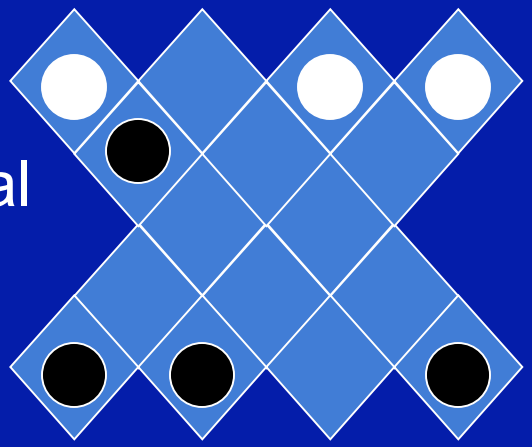
# Oska

- two players
- four pieces each
- piece movement:
  - piece may be moved one space forward on the diagonal
  - piece may jump forward on the diagonal over an opponent's piece to an empty space, thus capturing the opponent's piece



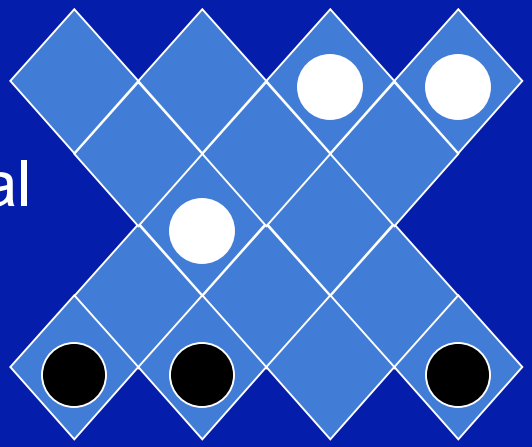
# Oska

- two players
- four pieces each
- piece movement:
  - piece may be moved one space forward on the diagonal
  - piece may jump forward on the diagonal over an opponent's piece to an empty space, thus capturing the opponent's piece



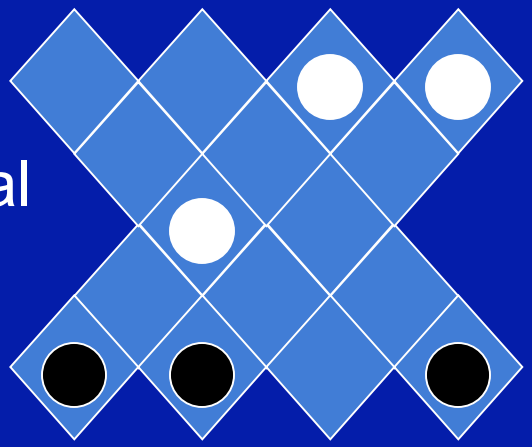
# Oska

- two players
- four pieces each
- piece movement:
  - piece may be moved one space forward on the diagonal
  - piece may jump forward on the diagonal over an opponent's piece to an empty space, thus capturing the opponent's piece



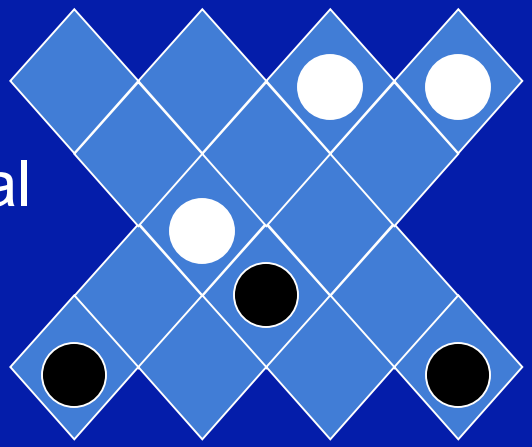
# Oska

- two players
- four pieces each
- piece movement:
  - piece may be moved one space forward on the diagonal
  - piece may jump forward on the diagonal over an opponent's piece to an empty space, thus capturing the opponent's piece
  - you don't have to capture if you don't want to



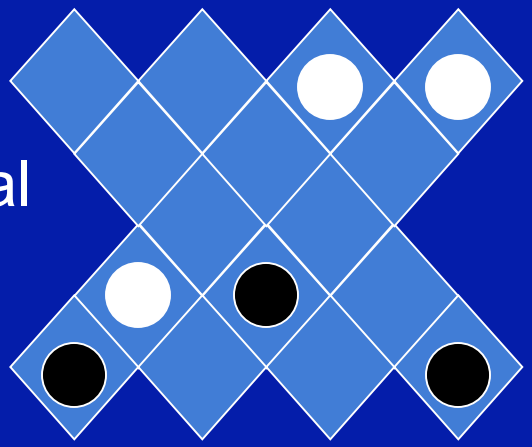
# Oska

- two players
- four pieces each
- piece movement:
  - piece may be moved one space forward on the diagonal
  - piece may jump forward on the diagonal over an opponent's piece to an empty space, thus capturing the opponent's piece
  - you don't have to capture if you don't want to



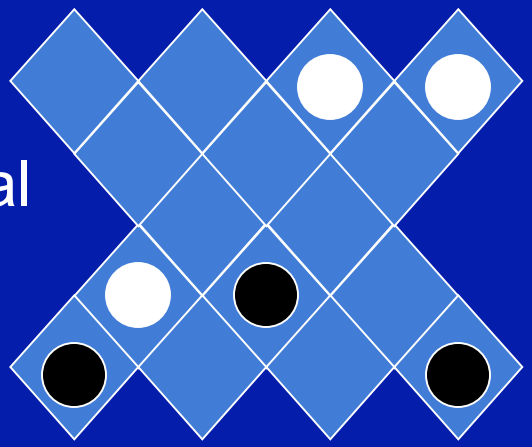
# Oska

- two players
- four pieces each
- piece movement:
  - piece may be moved one space forward on the diagonal
  - piece may jump forward on the diagonal over an opponent's piece to an empty space, thus capturing the opponent's piece
  - you don't have to capture if you don't want to



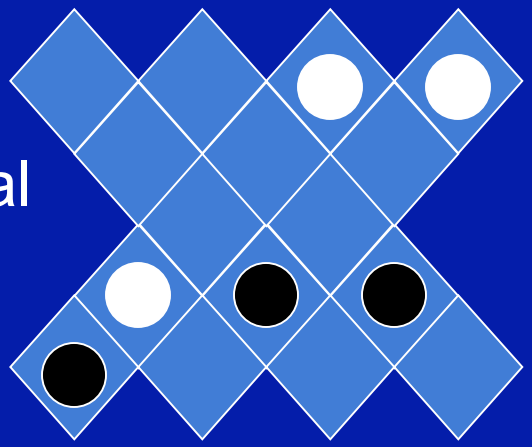
# Oska

- two players
- four pieces each
- piece movement:
  - piece may be moved one space forward on the diagonal
  - piece may jump forward on the diagonal over an opponent's piece to an empty space, thus capturing the opponent's piece
  - you don't have to capture if you don't want to
  - multiple jumps not permitted



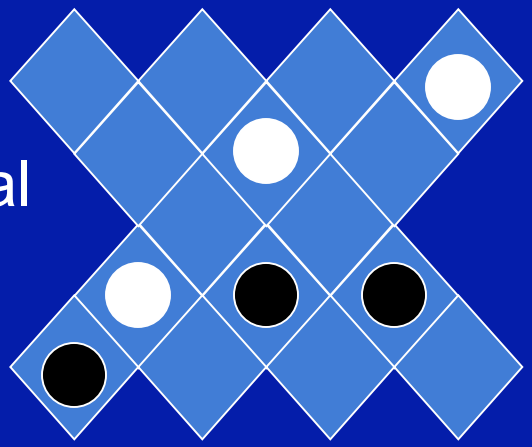
# Oska

- two players
- four pieces each
- piece movement:
  - piece may be moved one space forward on the diagonal
  - piece may jump forward on the diagonal over an opponent's piece to an empty space, thus capturing the opponent's piece
  - you don't have to capture if you don't want to
  - multiple jumps not permitted



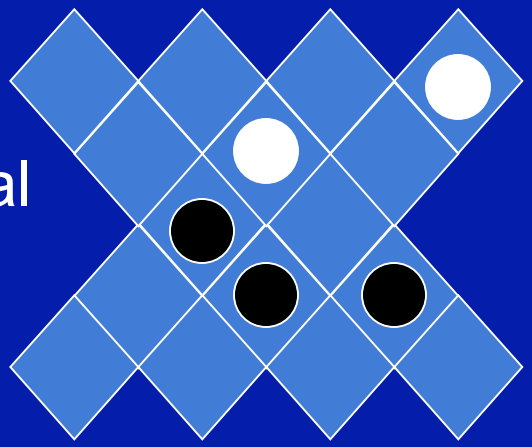
# Oska

- two players
- four pieces each
- piece movement:
  - piece may be moved one space forward on the diagonal
  - piece may jump forward on the diagonal over an opponent's piece to an empty space, thus capturing the opponent's piece
  - you don't have to capture if you don't want to
  - multiple jumps not permitted



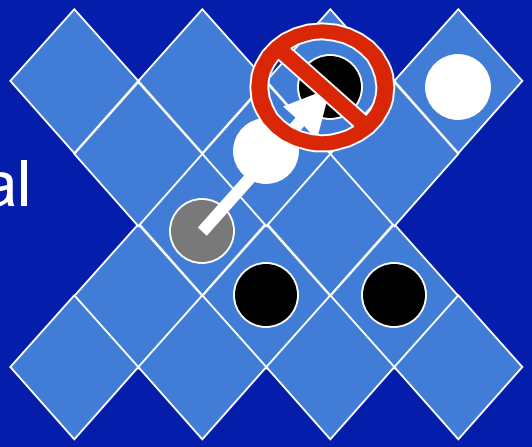
# Oska

- two players
- four pieces each
- piece movement:
  - piece may be moved one space forward on the diagonal
  - piece may jump forward on the diagonal over an opponent's piece to an empty space, thus capturing the opponent's piece
  - you don't have to capture if you don't want to
  - multiple jumps not permitted



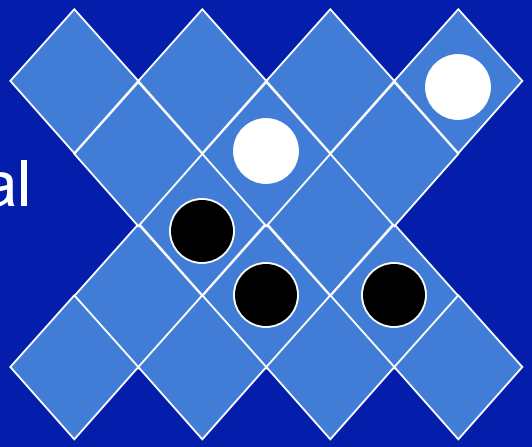
# Oska

- two players
- four pieces each
- piece movement:
  - piece may be moved one space forward on the diagonal
  - piece may jump forward on the diagonal over an opponent's piece to an empty space, thus capturing the opponent's piece
  - you don't have to capture if you don't want to
  - multiple jumps not permitted



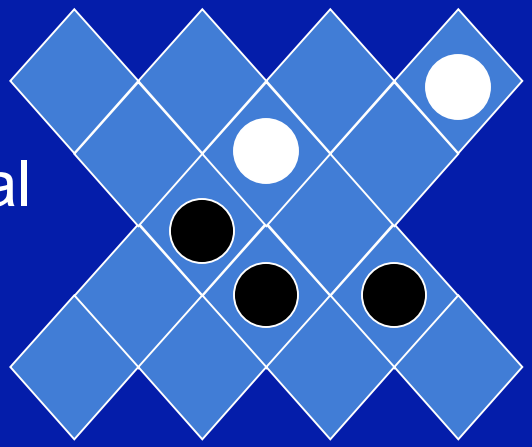
# Oska

- two players
- four pieces each
- piece movement:
  - piece may be moved one space forward on the diagonal
  - piece may jump forward on the diagonal over an opponent's piece to an empty space, thus capturing the opponent's piece
  - you don't have to capture if you don't want to
  - multiple jumps not permitted



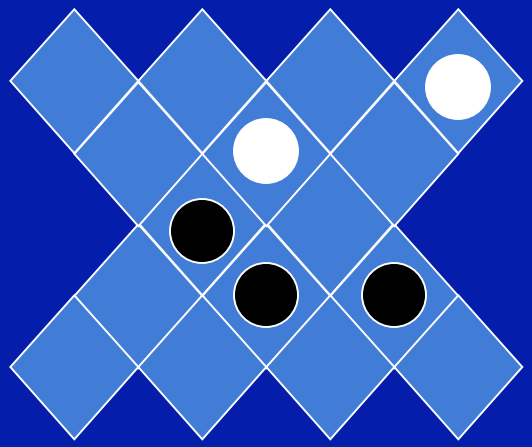
# Oska

- two players
- four pieces each
- piece movement:
  - piece may be moved one space forward on the diagonal
  - piece may jump forward on the diagonal over an opponent's piece to an empty space, thus capturing the opponent's piece
  - you don't have to capture if you don't want to
  - multiple jumps not permitted
  - if a player can't make a legal move on a turn, the player loses turn and opponent moves again



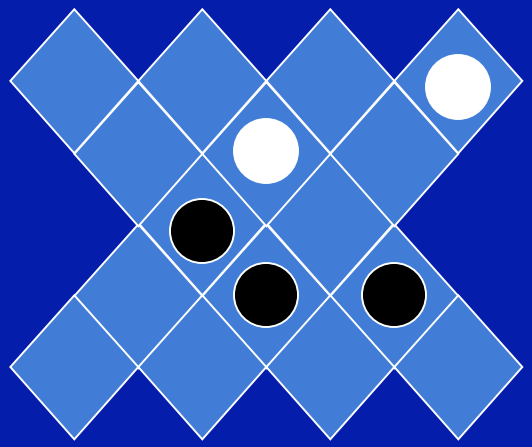
# Oska

- how to win



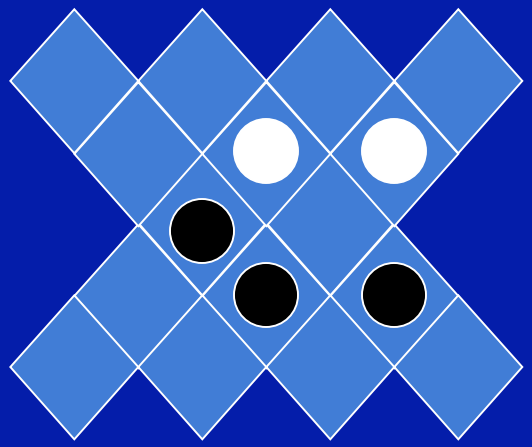
# Oska

- how to win
  - capture all your opponent's pieces



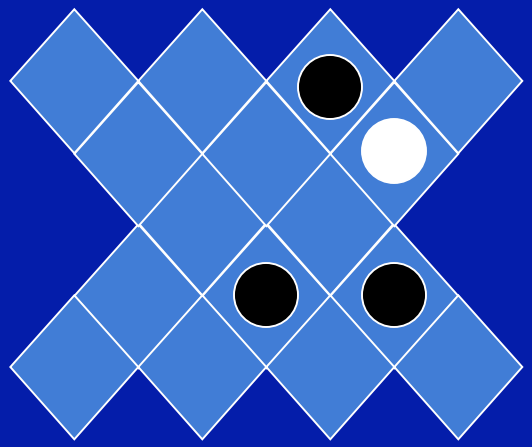
# Oska

- how to win
  - capture all your opponent's pieces



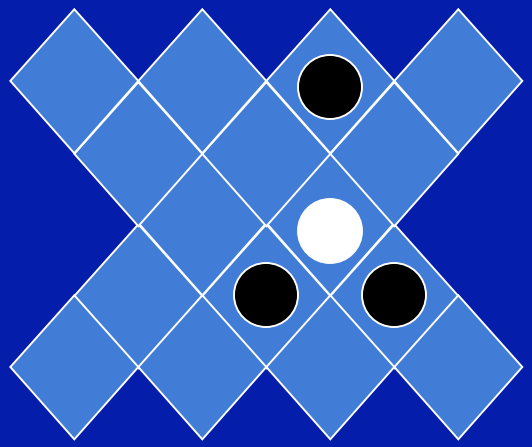
# Oska

- how to win
  - capture all your opponent's pieces



# Oska

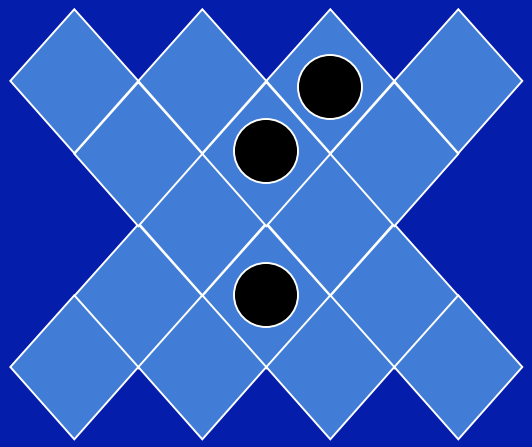
- how to win
  - capture all your opponent's pieces



# Oska

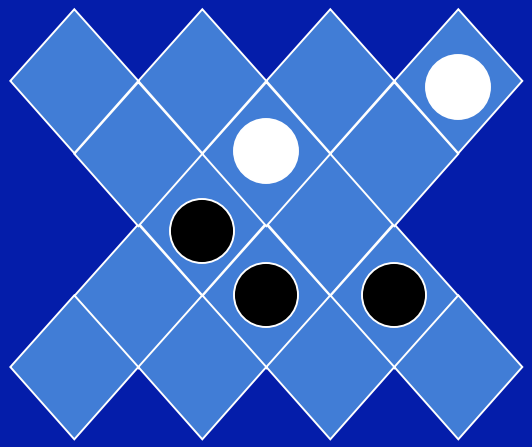
- how to win
  - capture all your opponent's pieces

black wins



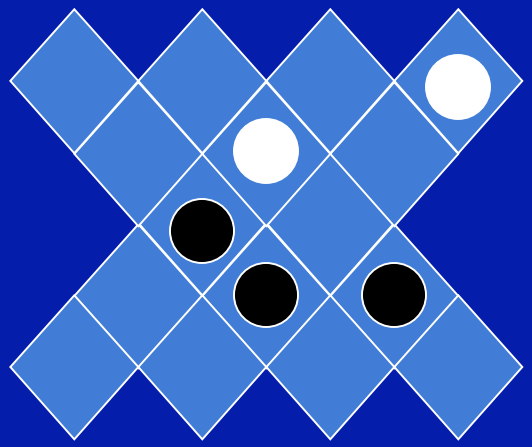
# Oska

- how to win
  - capture all your opponent's pieces



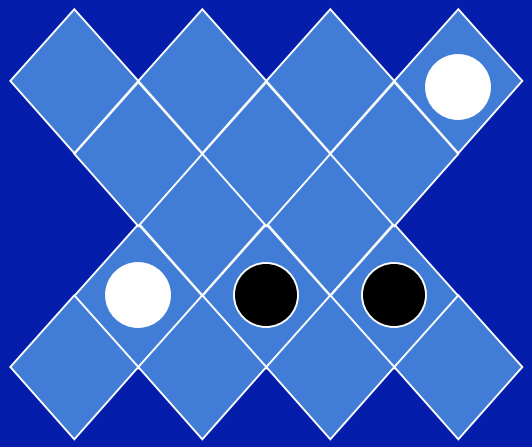
# Oska

- how to win
  - capture all your opponent's pieces
  - move all your remaining pieces to your opponent's starting row



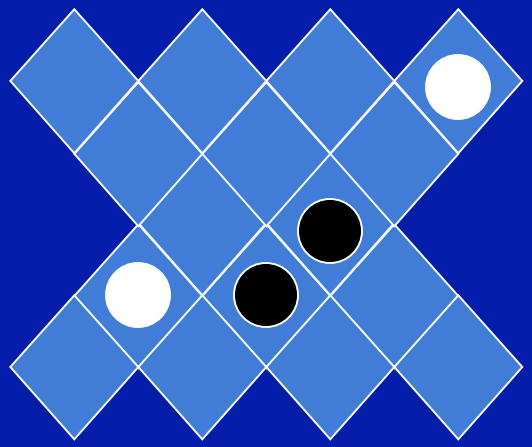
# Oska

- how to win
  - capture all your opponent's pieces
  - move all your remaining pieces to your opponent's starting row



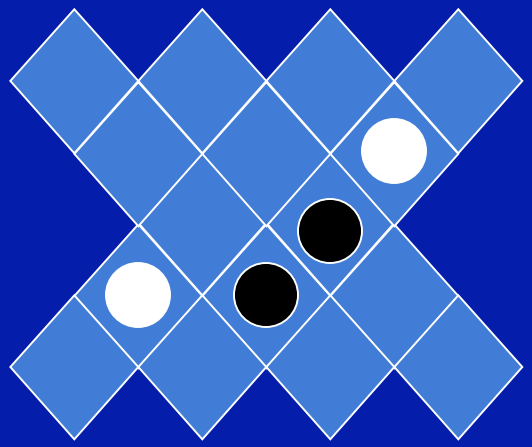
# Oska

- how to win
  - capture all your opponent's pieces
  - move all your remaining pieces to your opponent's starting row



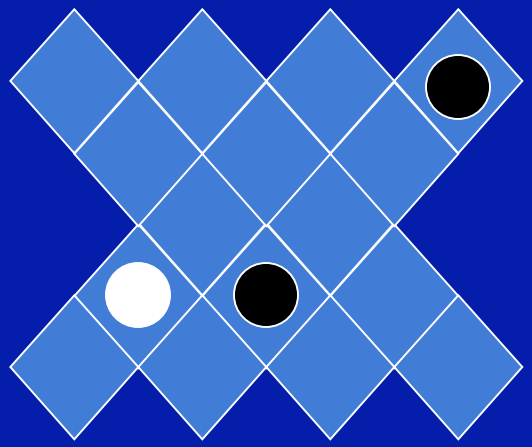
# Oska

- how to win
  - capture all your opponent's pieces
  - move all your remaining pieces to your opponent's starting row



# Oska

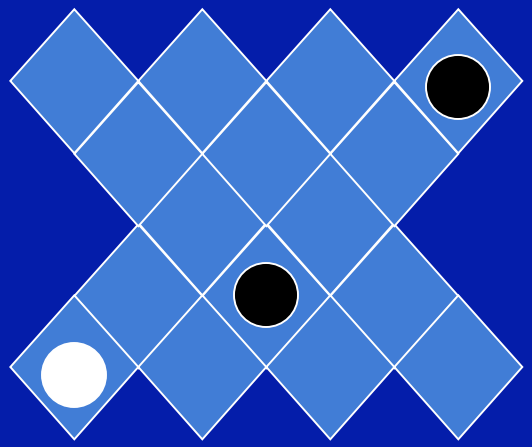
- how to win
  - capture all your opponent's pieces
  - move all your remaining pieces to your opponent's starting row



# Oska

- how to win
  - capture all your opponent's pieces
  - move all your remaining pieces to your opponent's starting row

white wins



# What does this have to do with you? Consider this predicate:

```
oska([[w w w w][0 0 0][0 0][0 0 0][b b b b]],
     w,
     3,
     [[w w w 0][0 0 w][0 0][0 0 0][b b b b]]).
```

This predicate is true if

- the first argument is a legal board in an Oska game
- the second argument indicates the colour of the pieces being “moved” by the oska predicate
- the third argument indicates how deep (number of moves/plies/levels) the game search goes
- the fourth argument is the best move that the oska predicate can find using its move generator, board evaluation function, and minimax algorithm

# Welcome to your term project

```
oska([[w w w w][0 0 0][0 0][0 0 0][b b b b]],
     w,
     3,
     [[w w w 0][0 0 w][0 0][0 0 0][b b b b]]).
```

- due 6:00am Monday, November 29, 2004
- you can work individually or in pairs
  but I need to know who the pairs no later than this Sunday, October 31
- more details will be posted this week, including documentation requirements

# One other thing....

```
oska([[w w w w][0 0 0][0 0][0 0 0][b b b b]],
     w,
     3,
     [[w w w 0][0 0 w][0 0][0 0 0][b b b b]]).
```

could look like this instead:

```
oska([[w w w w w][0 0 0 0][0 0 0][0 0]
      [0 0 0][0 0 0 0][b b b b b]],
     w,
     3,
     [[w w w w 0][0 0 0 w][0 0 0][0 0]
      [0 0 0][0 0 0 0][b b b b b]]).
```

# Your program vs. Deep Blue

- game search
  - move generator
  - static board evaluator
  - minimax algorithm
  - pruning (maybe)
- one processor

- CILOG

- grit and determination

- game search
  - move generator
  - static board evaluator
  - minimax algorithm
  - pruning
- 480 custom-fabricated chess-playing chips
- algorithms and representations hardwired on chips
- selective deepening by reallocating chips on the fly
- giant libraries of openings, endgames, and championship matches

# Back to heuristic search techniques

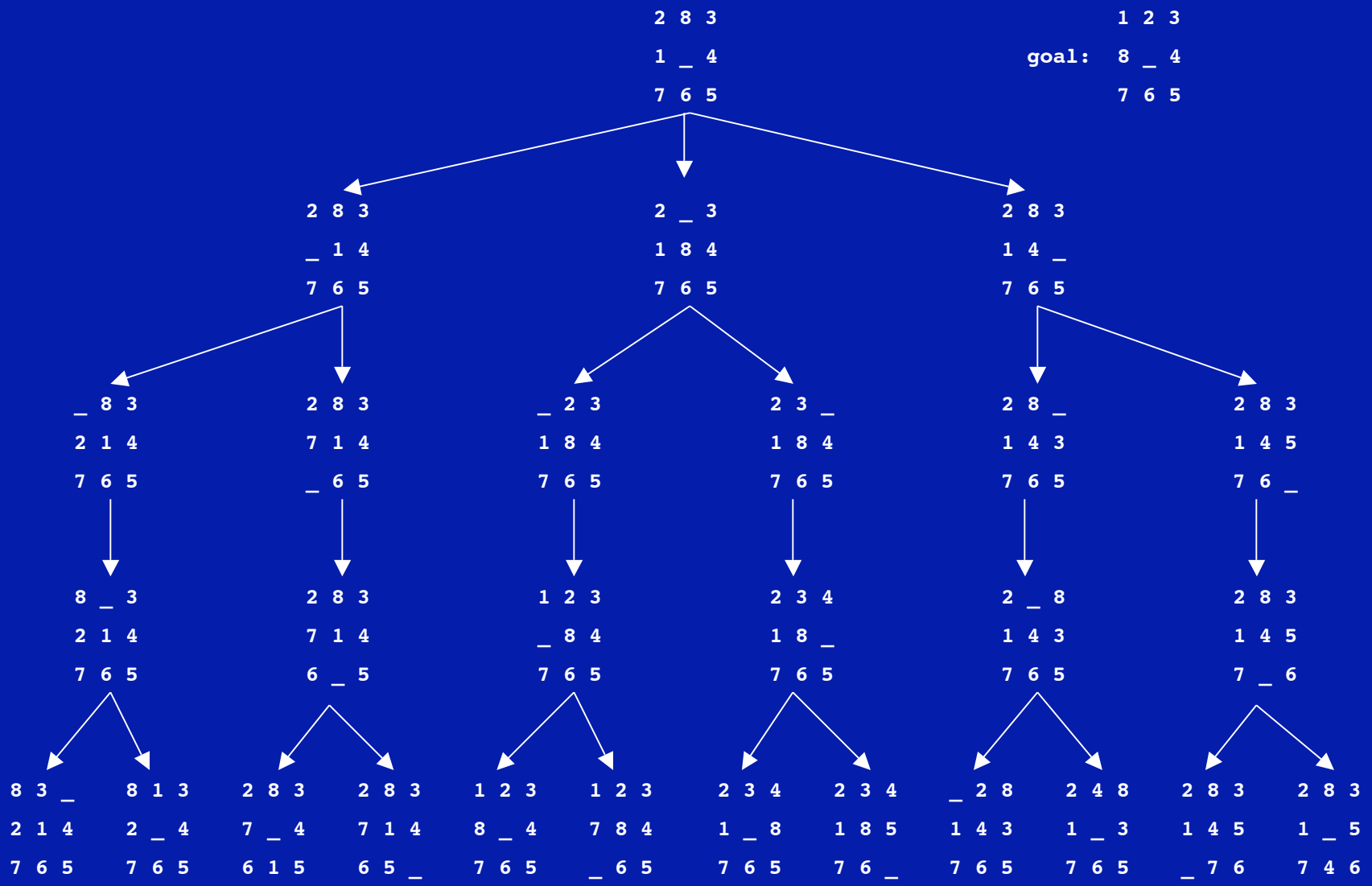
# Heuristic depth-first search algorithm

Given a set of start nodes, a set of goal nodes,
and a graph (i.e., the nodes and arcs):

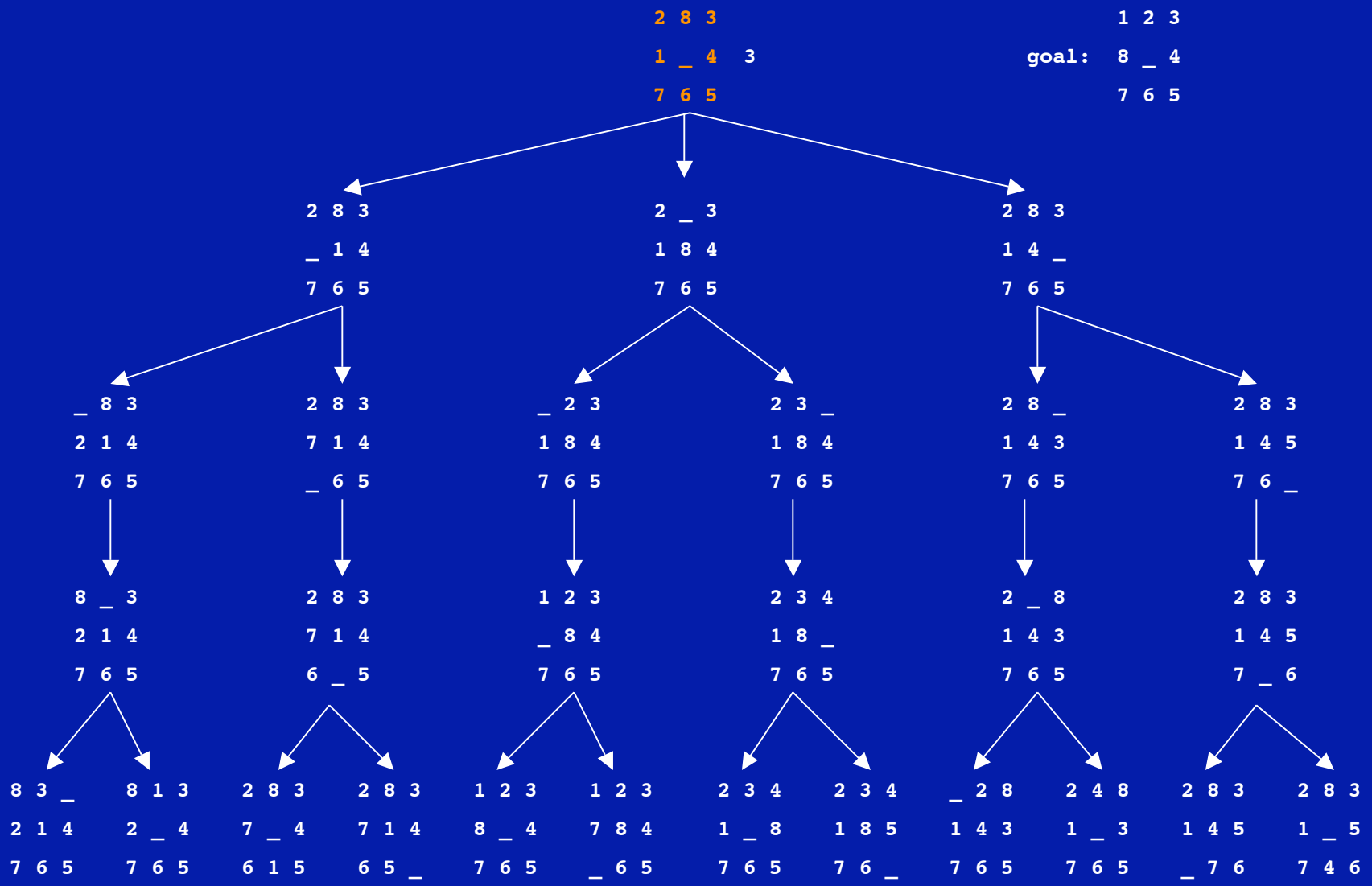
apply heuristic h(n) to start nodes
make a “list” of the start nodes - let’s call it the “frontier”
sort the frontier by h(n) values

repeat
  if no nodes on the frontier then terminate with failure
  choose one node from the front of the frontier and remove it
  if the chosen node matches the goal node
    then terminate with success
    else get next nodes (neighbors) and h(n) values
      and sort those nodes by h(n) values
      and put those sorted nodes on the front of the frontier
end repeat

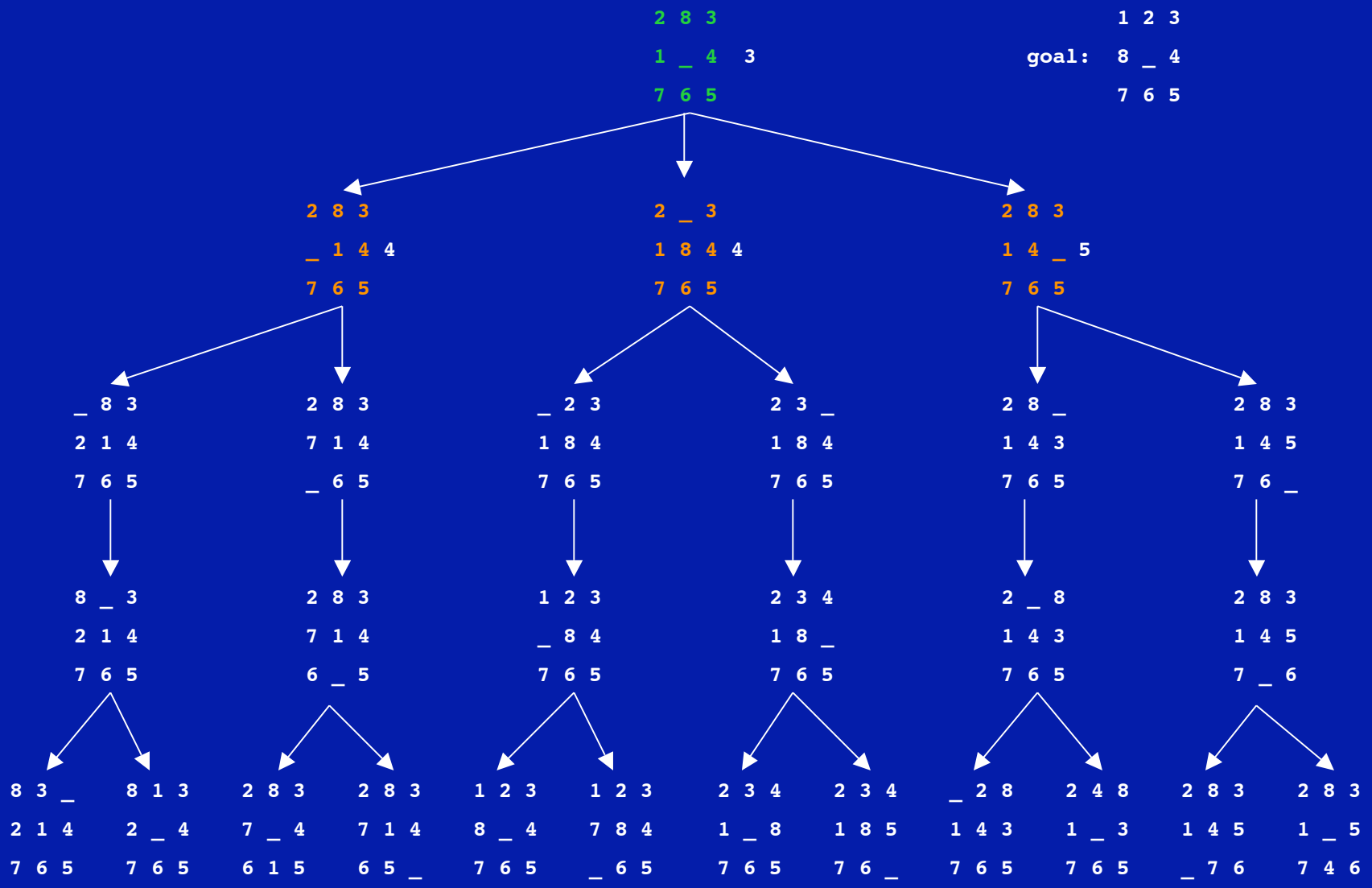
# Heuristic depth-first search - tiles out of place



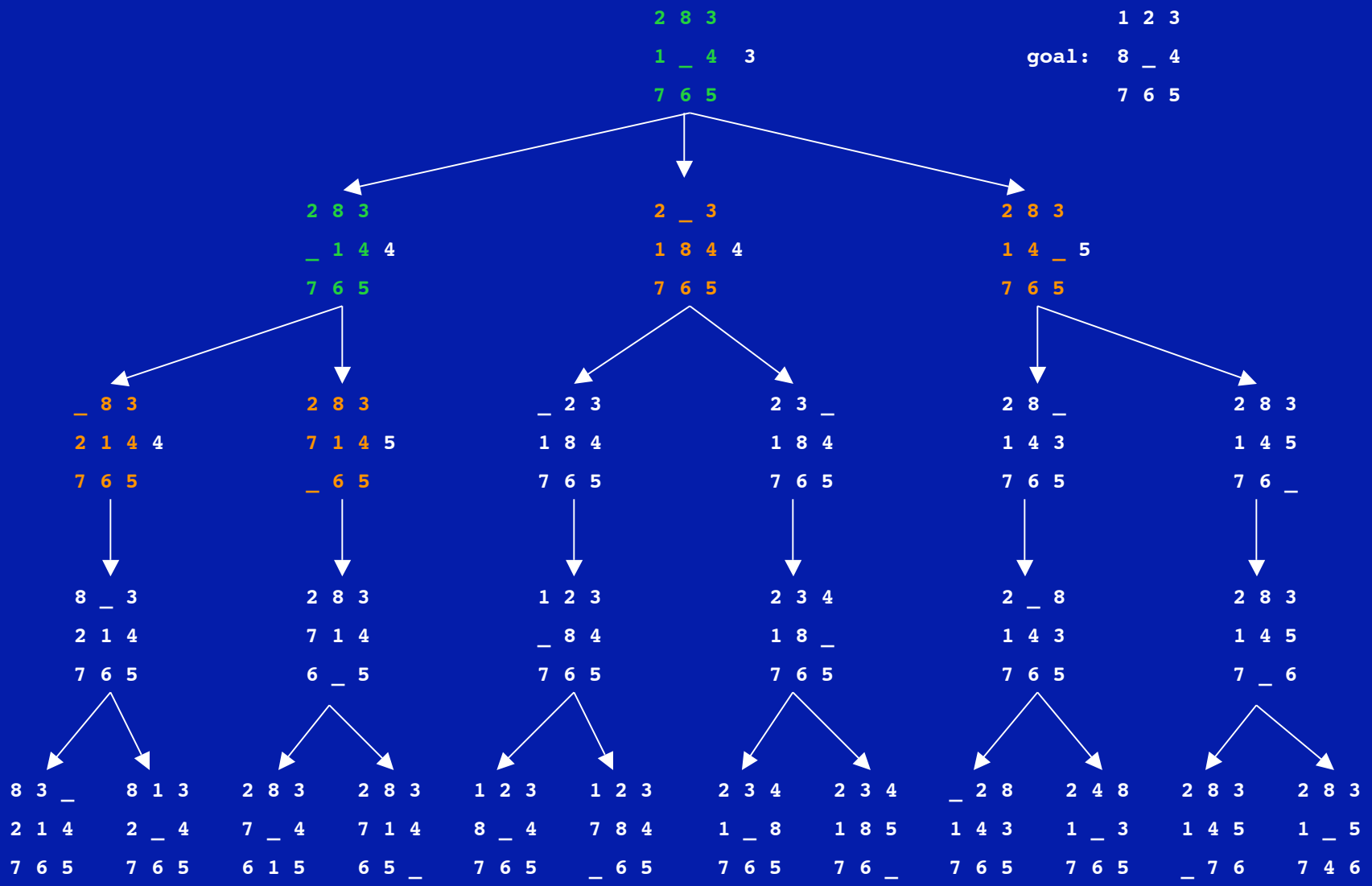
# Heuristic depth-first search - tiles out of place



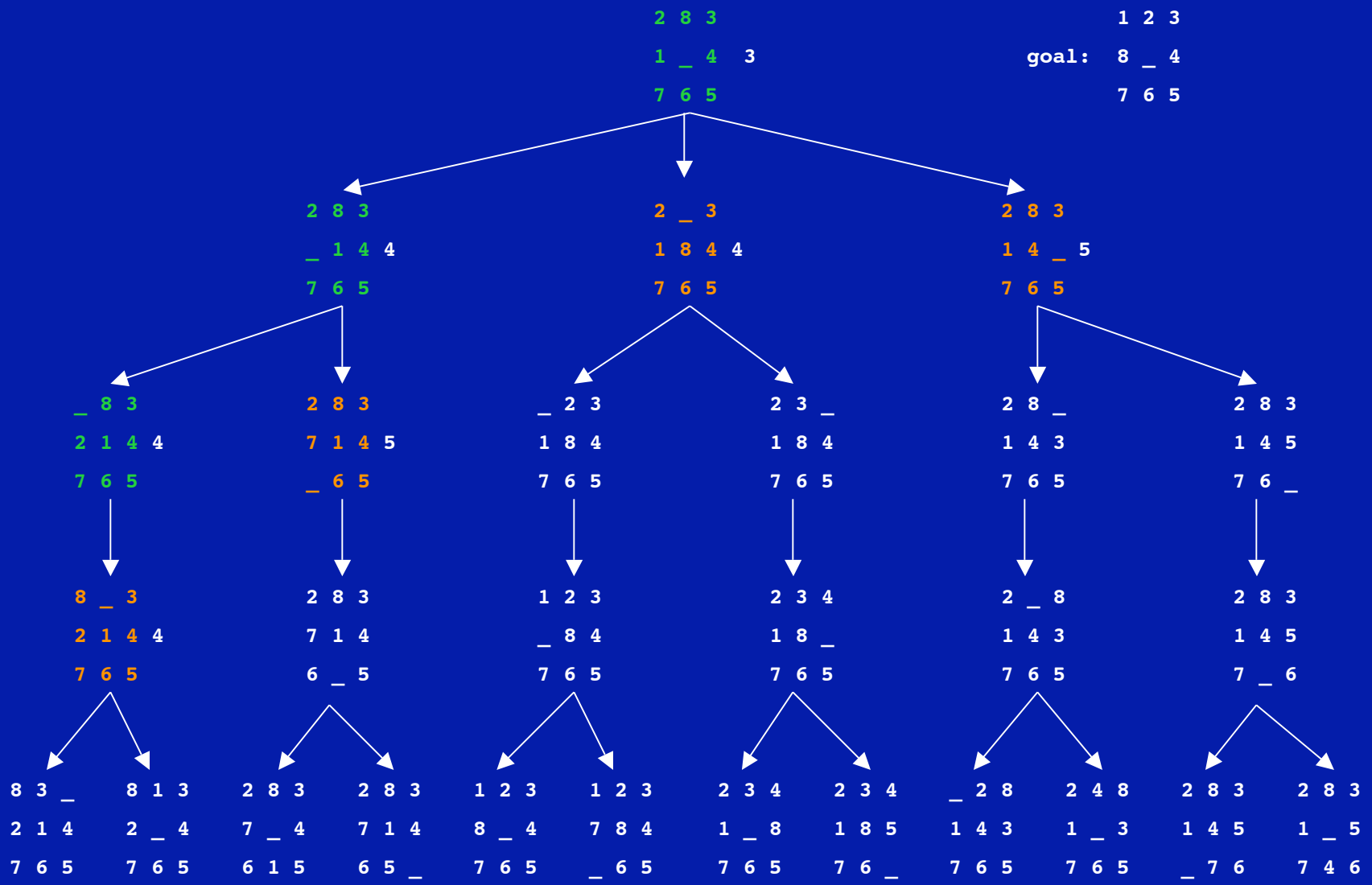
# Heuristic depth-first search - tiles out of place



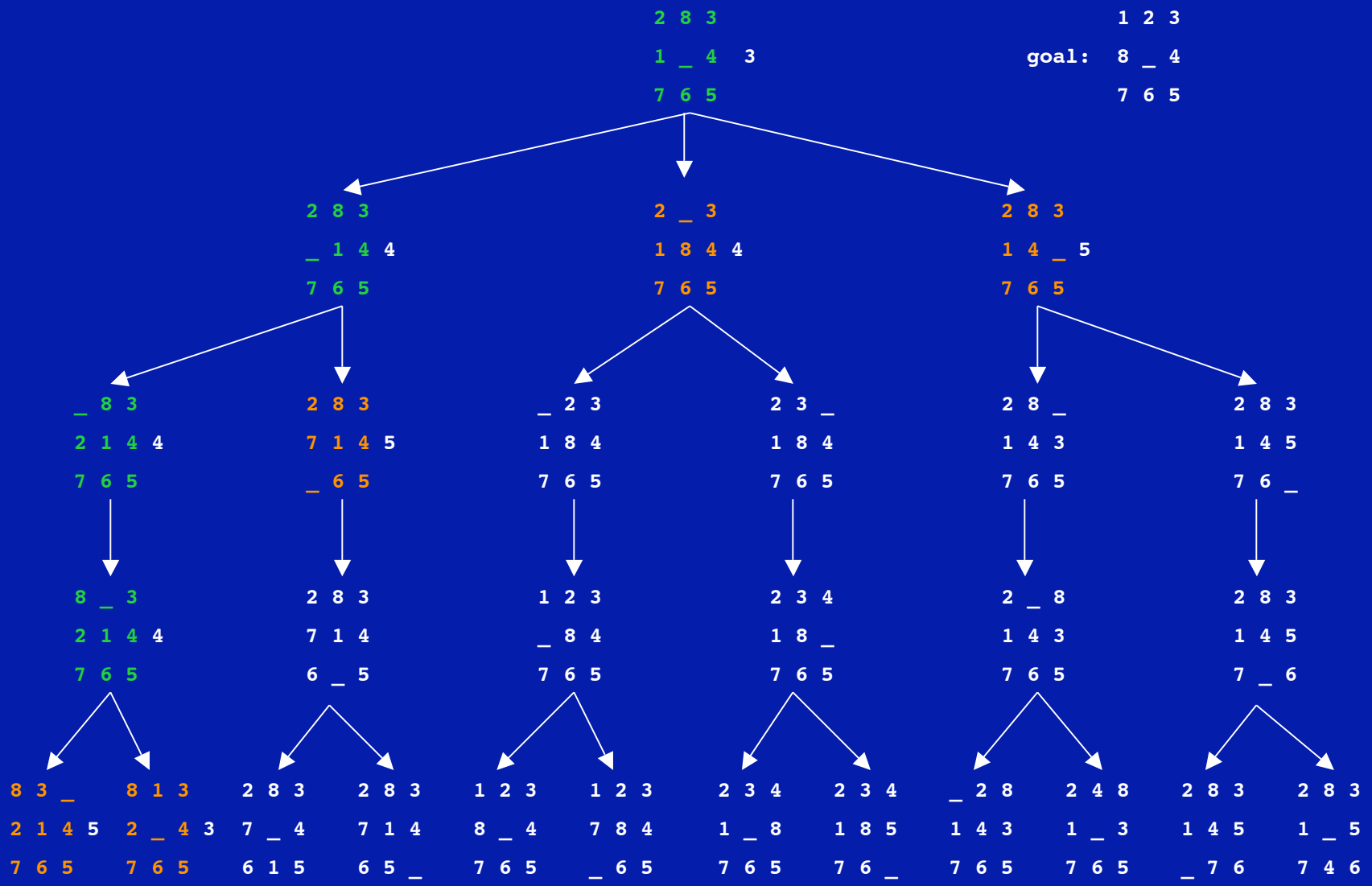
# Heuristic depth-first search - tiles out of place



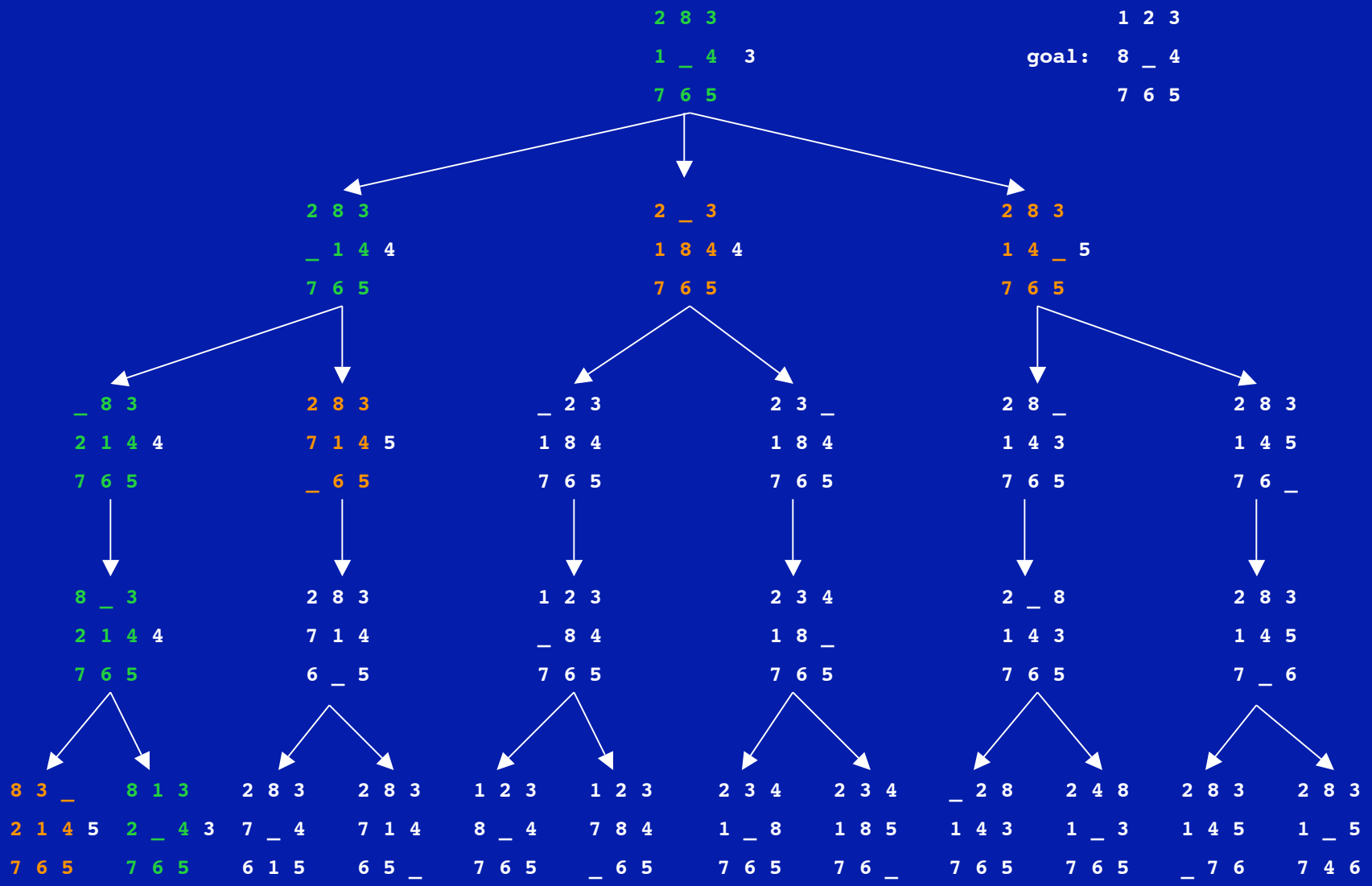
# Heuristic depth-first search - tiles out of place



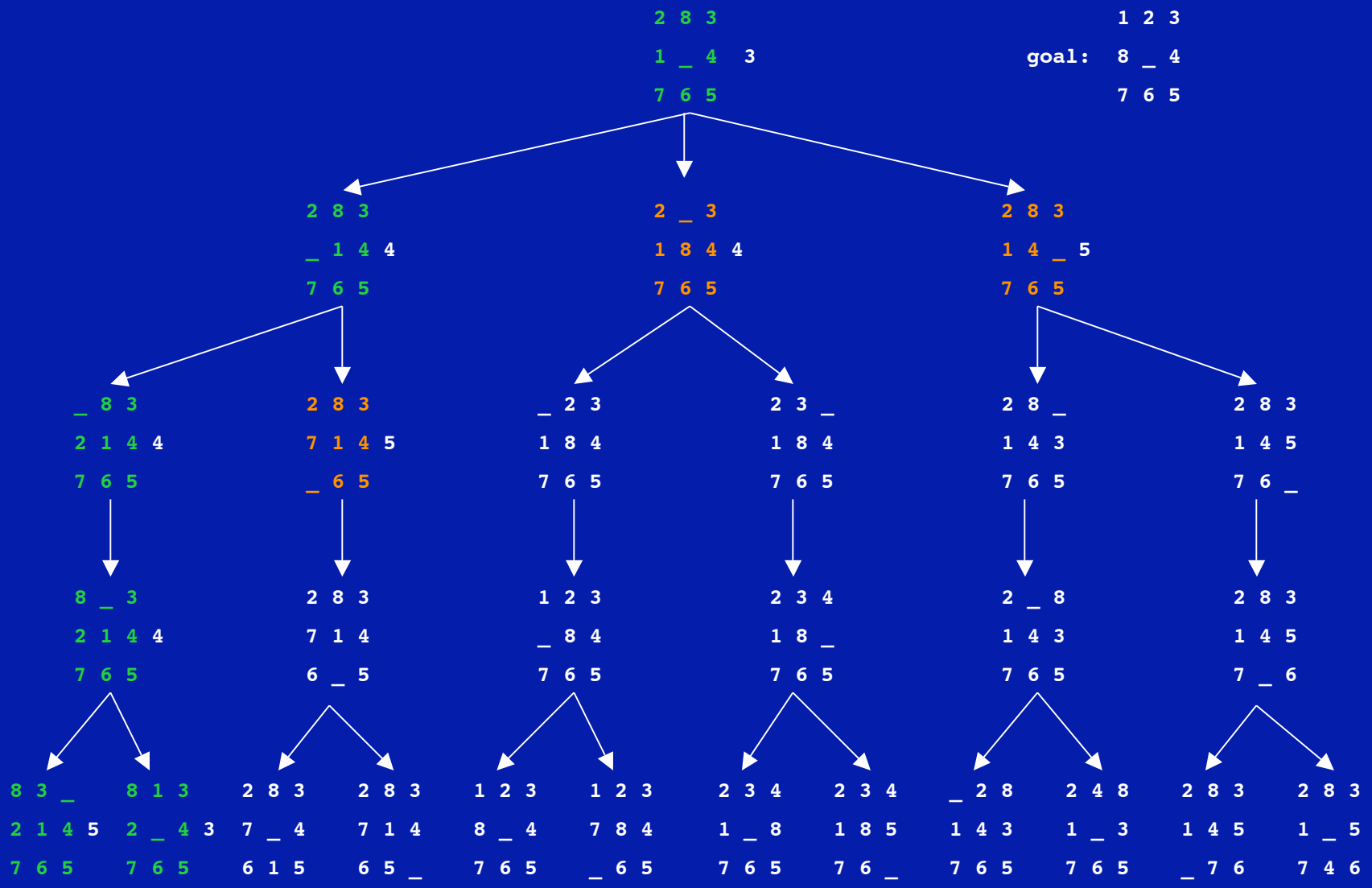
# Heuristic depth-first search - tiles out of place



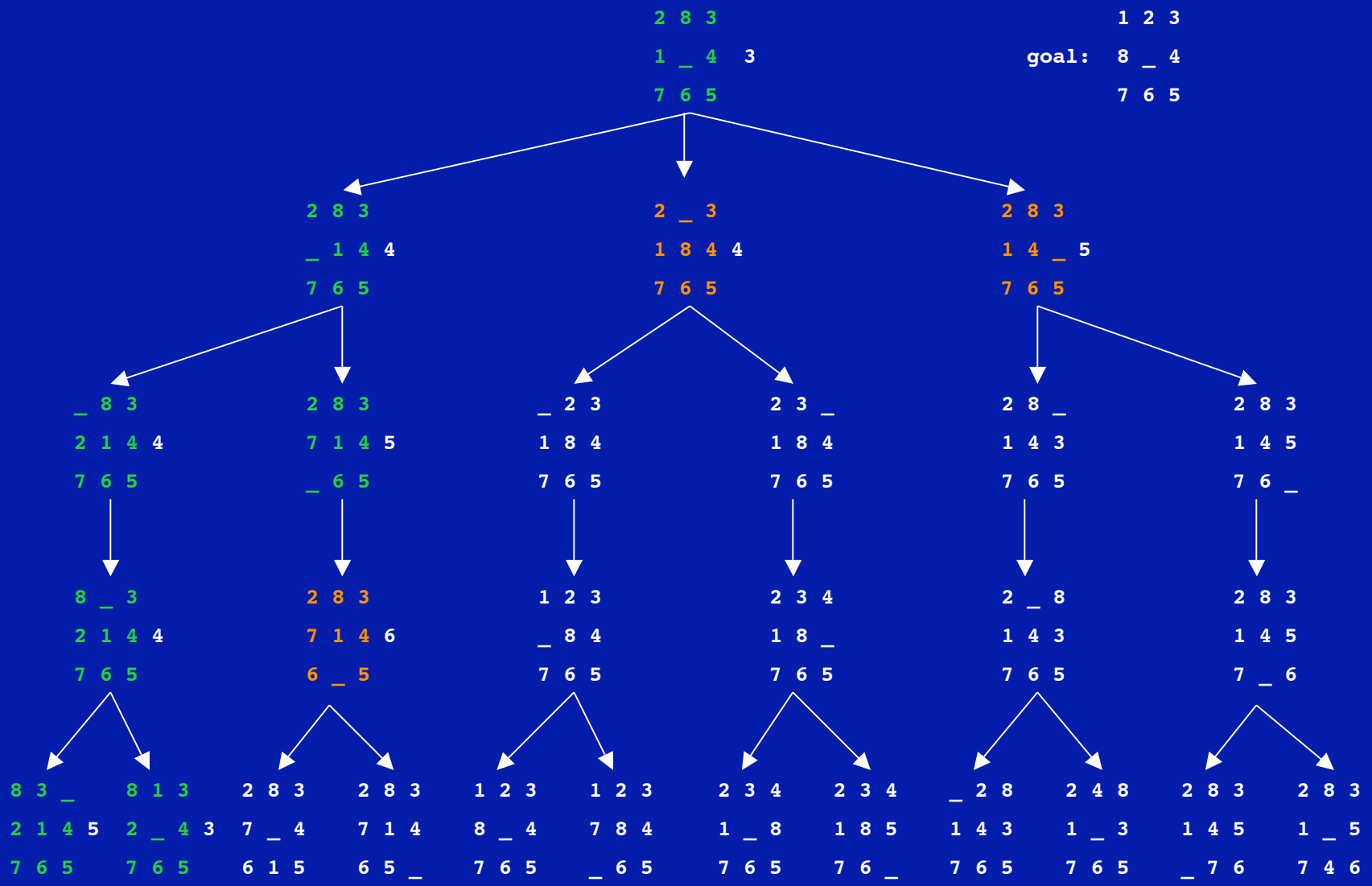
# Heuristic depth-first search - tiles out of place



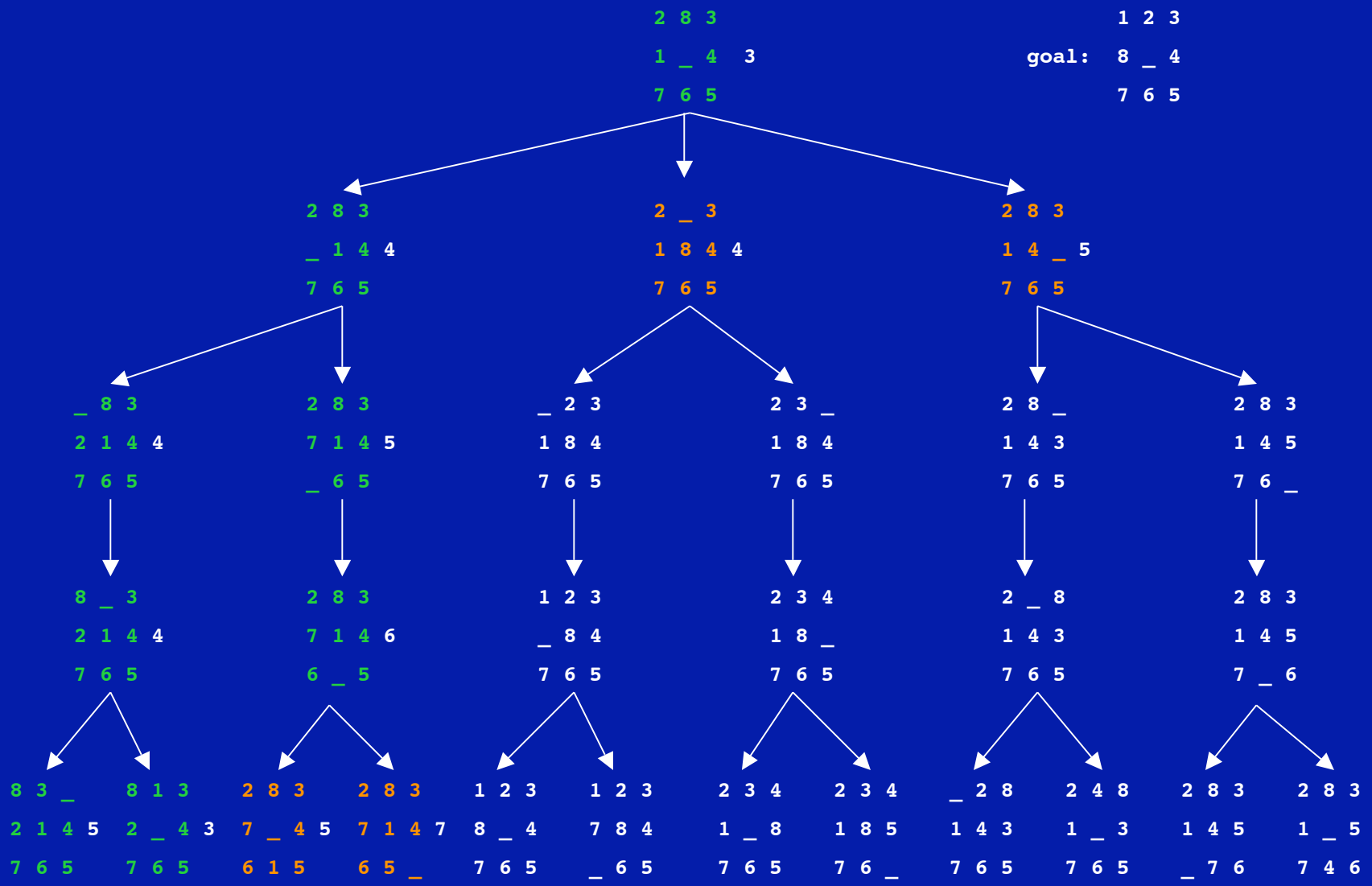
# Heuristic depth-first search - tiles out of place



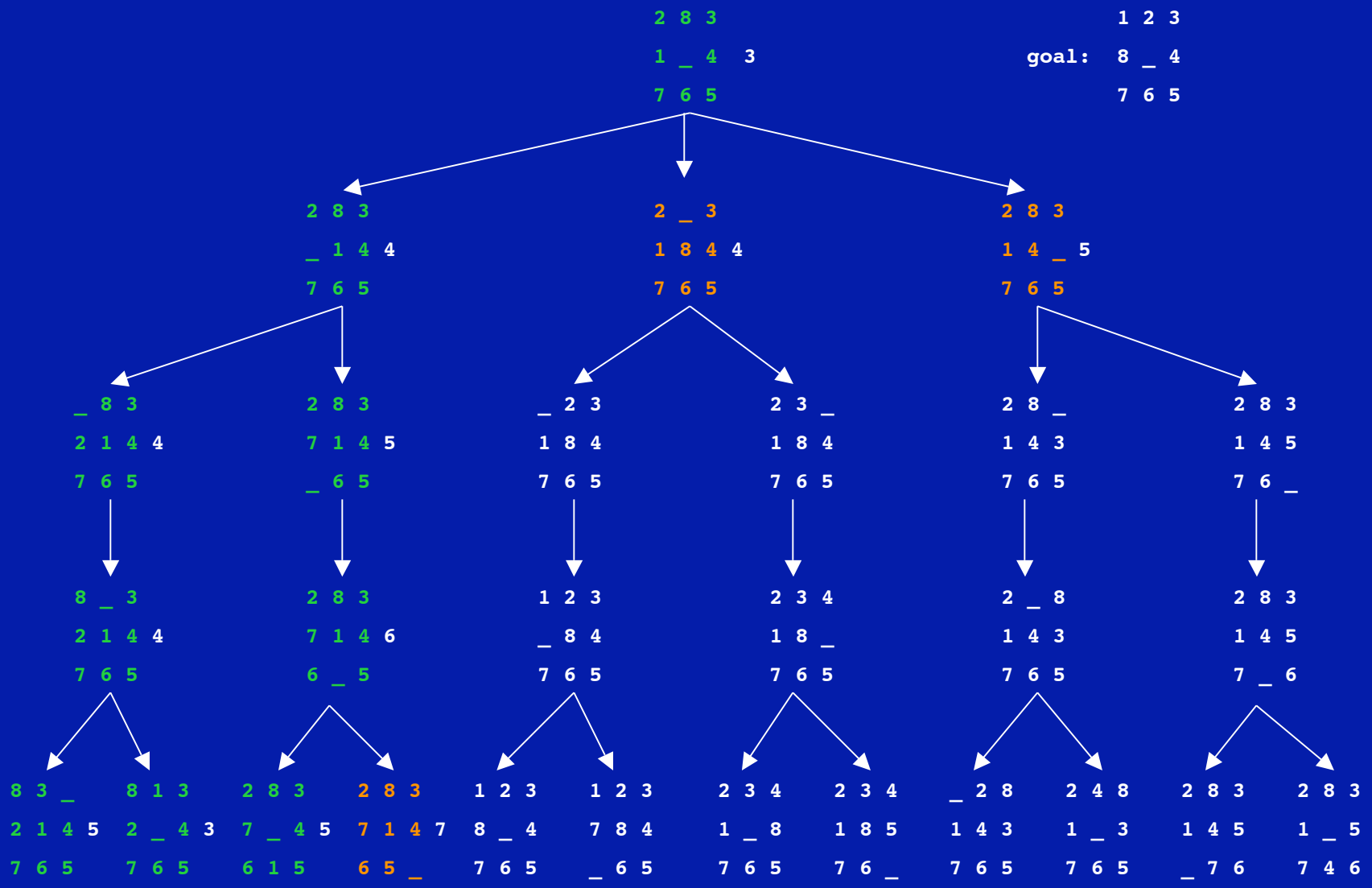
# Heuristic depth-first search - tiles out of place



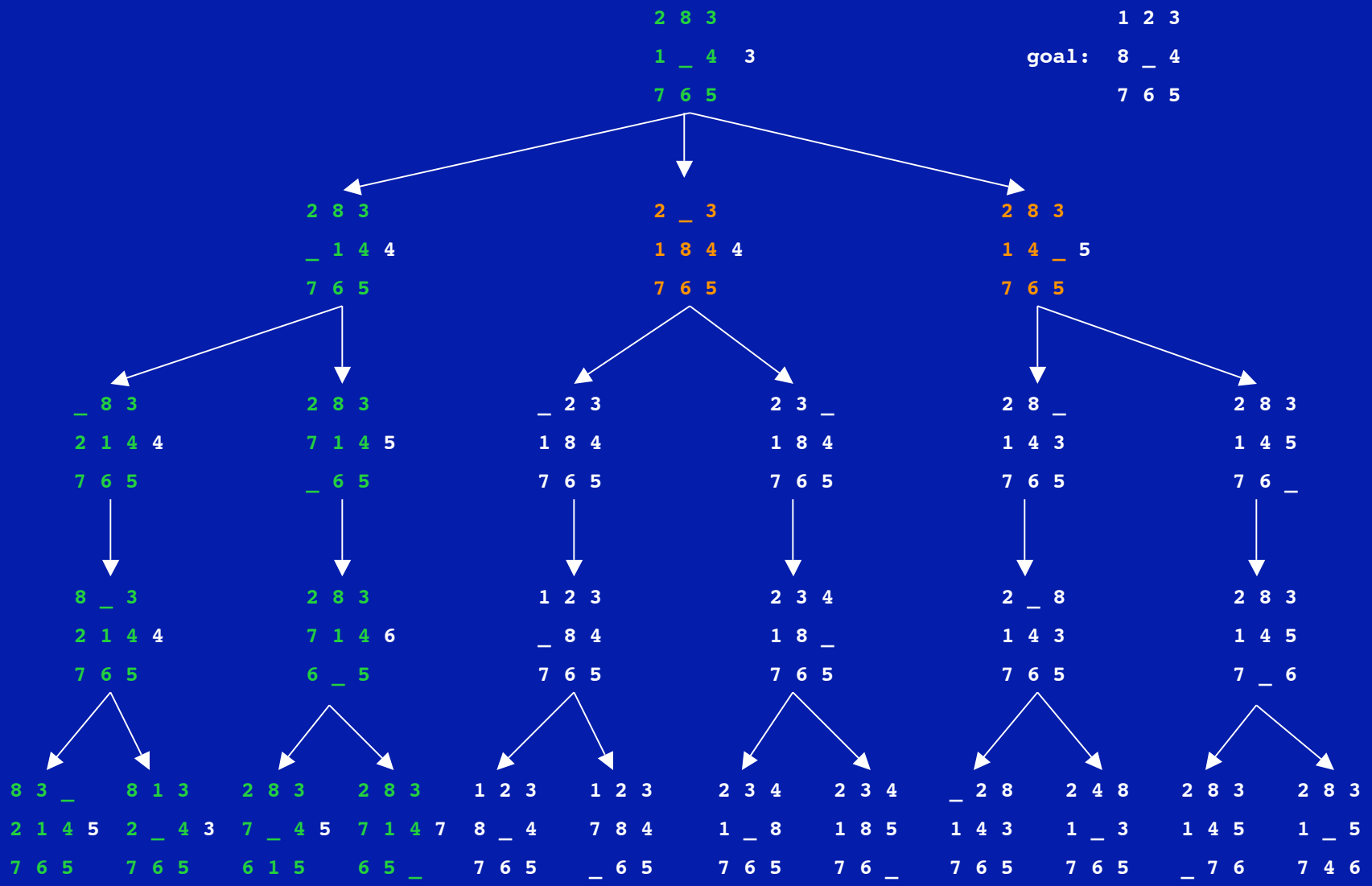
# Heuristic depth-first search - tiles out of place



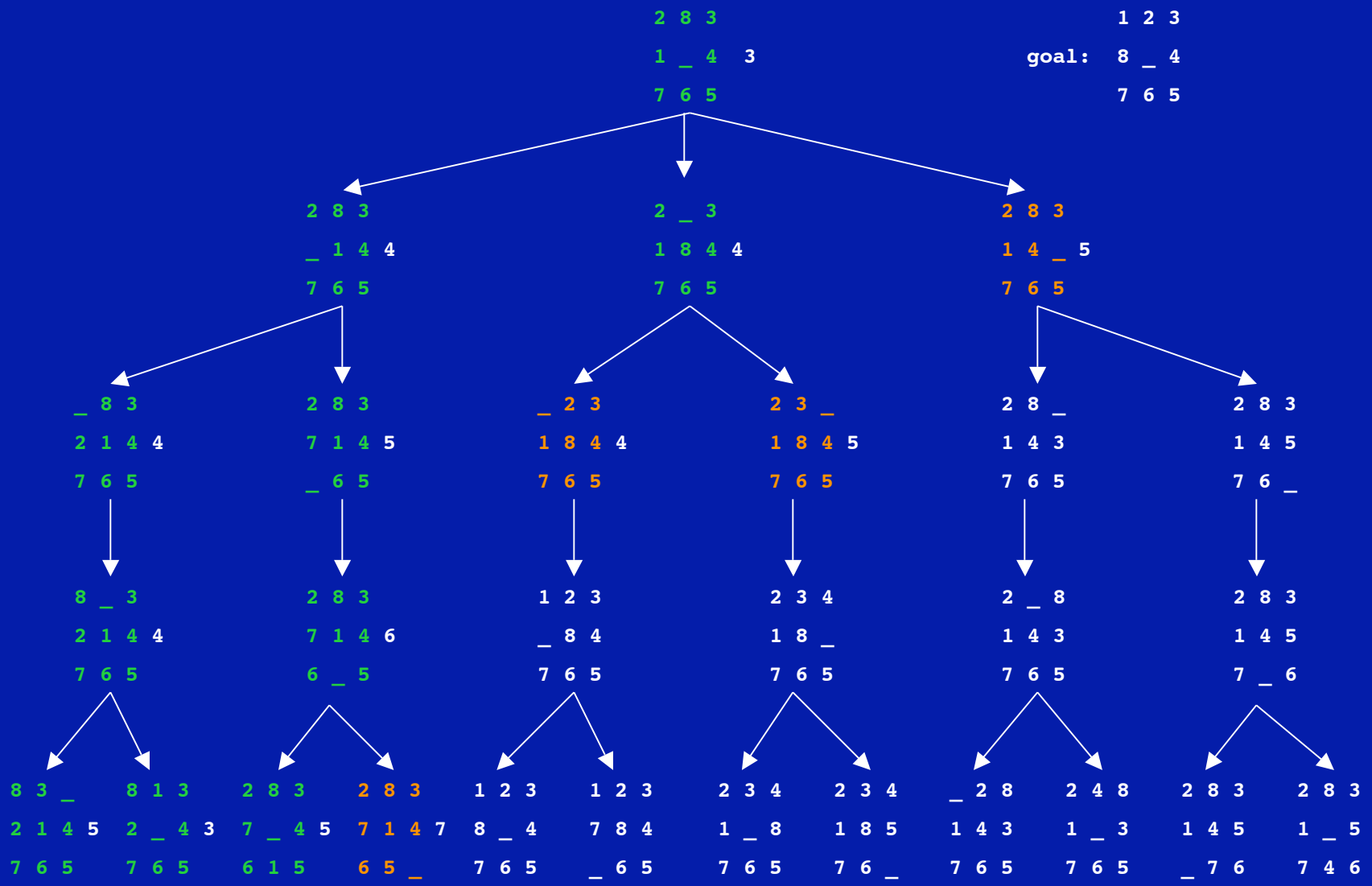
# Heuristic depth-first search - tiles out of place



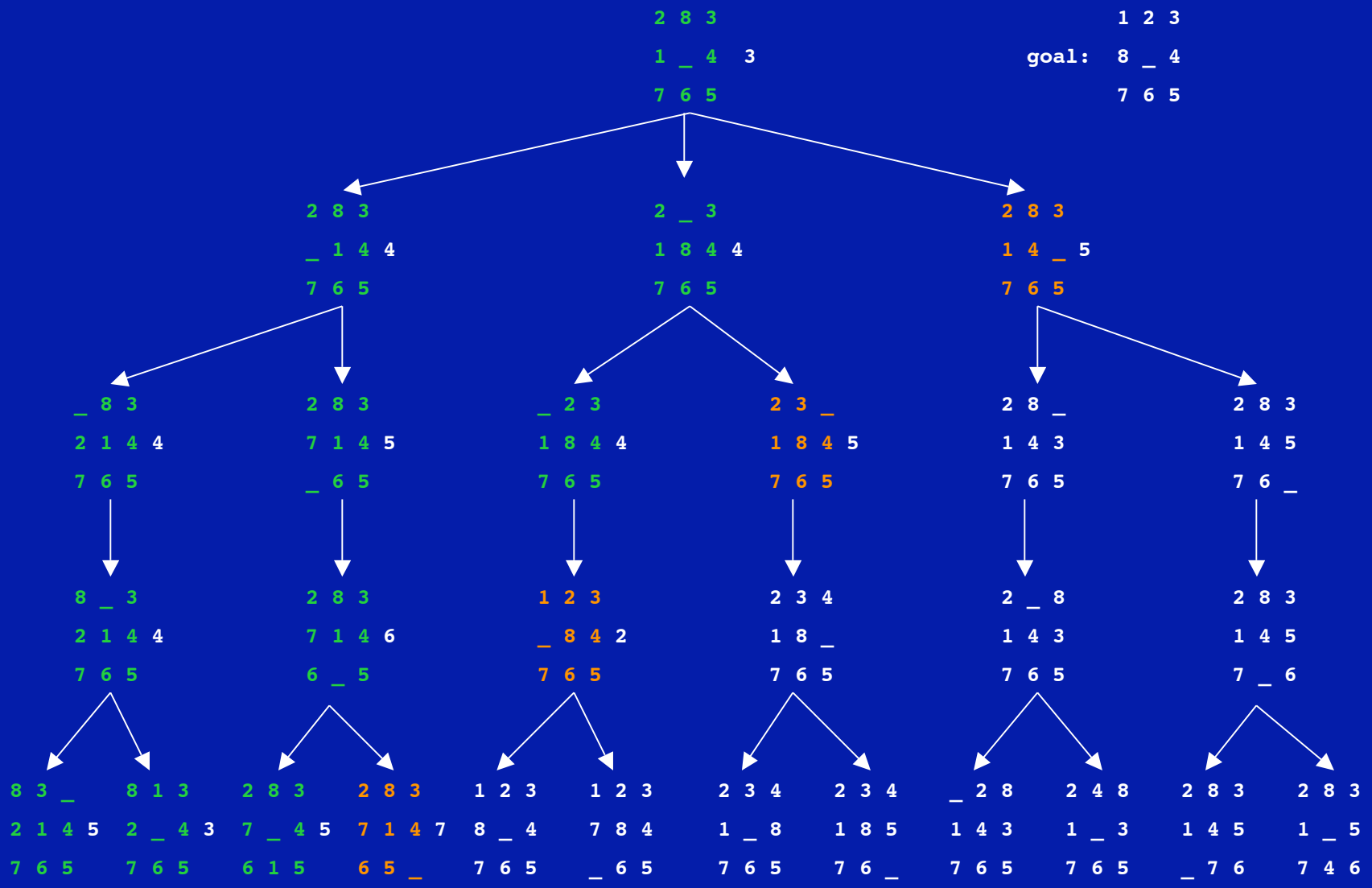
# Heuristic depth-first search - tiles out of place



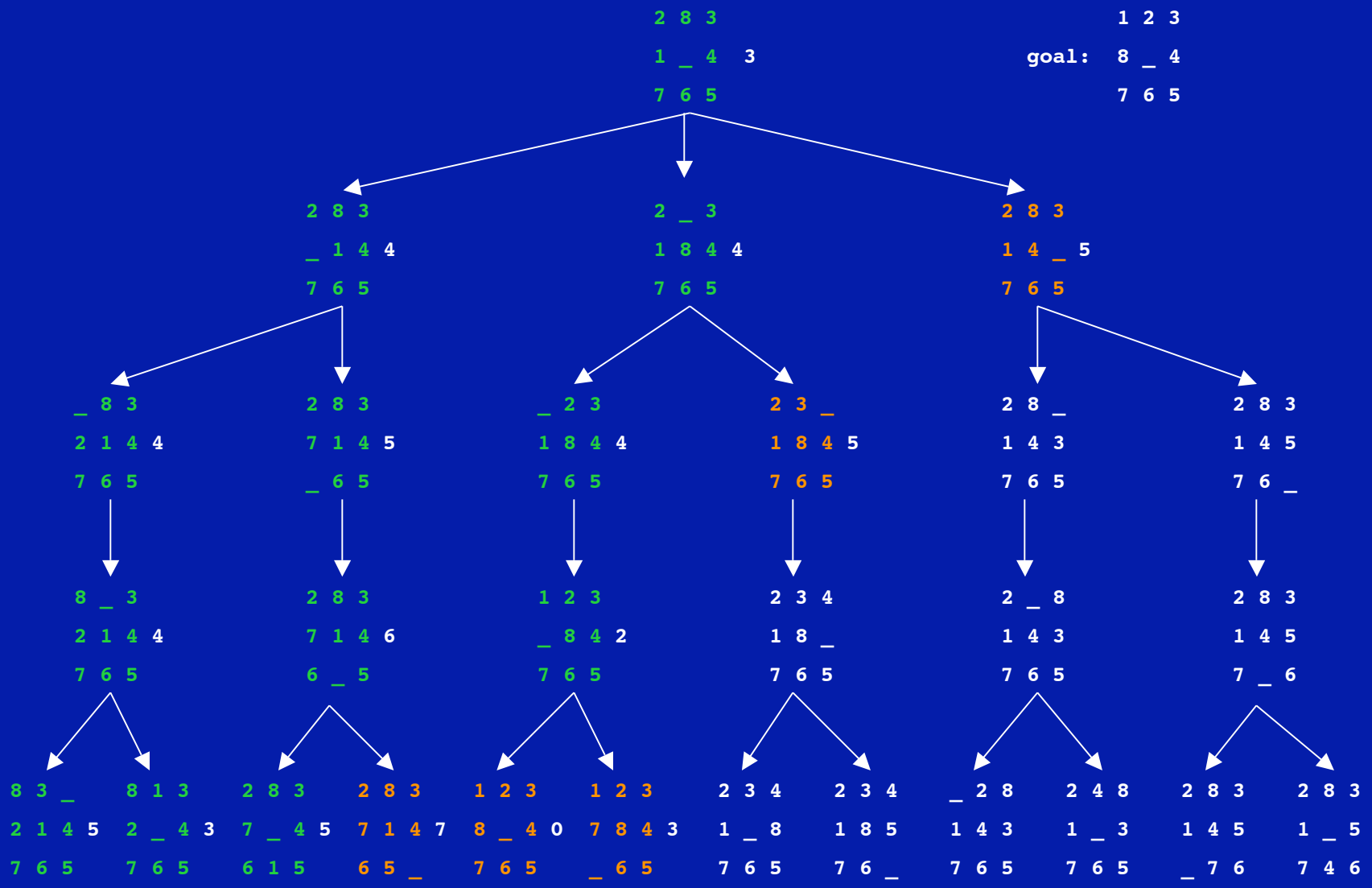
# Heuristic depth-first search - tiles out of place



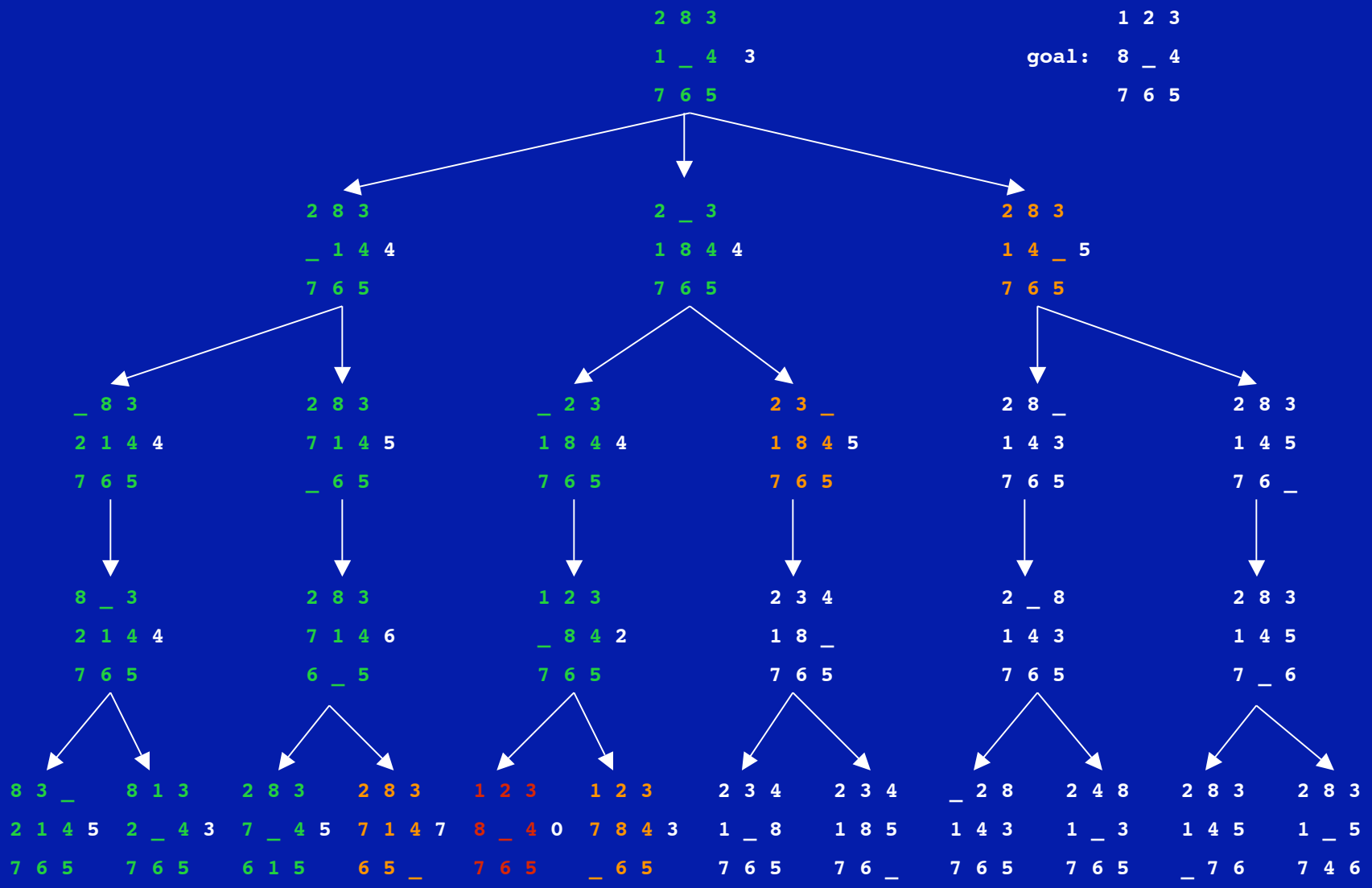
# Heuristic depth-first search - tiles out of place



# Heuristic depth-first search - tiles out of place



# Heuristic depth-first search - tiles out of place



# Why heuristic depth-first search?

# Comparing search strategies

| strategy | selection | halts? | space | optimal path |
|---|---|---|---|---|
| depth-first | last node added | no | linear | no |
| breadth-first | first node added | yes | exponential | yes* |
| lowest-cost first | minimal g(n) | yes | exponential | yes |
| best-first | globally minimal h(n) | no | exponential | no |
| heuristic depth-first | locally minimal h(n) | no | linear | no |

* assuming all arcs have the same cost

# What's missing?

A heuristic search strategy that is guaranteed to find the optimal (lowest-cost) path from start node to goal node

What do the strategies that find optimal paths have in common?

# Questions?