# Arithmetic Series Heuristic

**Problem 2**

**Part A**

The function 'jump-heuristic' accepts a jump-state as its input and returns a non-negative integer as the estimate for the number of steps required to reach the goal state from the given jump-state. The jump-state is a list of 3 non-negative integers N, pos and m, where N is the total length of the course, pos is the current position of the agent and m is the momentum of the agent. Notice that for a jump-state `(list N pos m)`, the agent has to travel a distance of `(- N pos)` to reach goal state. Recall that the agent can only slow down by a speed of 1 at every step. This means that for the agent to have a momentum of 1 at the last step, the agent must take at least `(- m 1)` steps, slowing down by 1 at each step. These two insights are the heart of `jump-heuristic`.

`jump-heuristic` employs *backward indcution* as it calculates the optimal number of steps from the goal state instead of from pos. The most distance that the agent can cover in its last step is 1, since the agent cannot have a higher momentum that 1 when it is in the goal state. This implies that the most distance the agent can cover in its last 2 steps is `(+ 1 2)`, as jumping by a step larger than 2 in the penultimate step means that the agent cannot reach the goal state with a momentum of less that 1. Continuing this *backward chaining*, the most distance that the agent can cover in $n$ steps is given by the arithmetic series `(+ 1 2 ... n)`, which adds up to $\frac{n(n+1)}{2}$. We equate the maximum distance that the agent can travel in $n$ steps with the distance the the agent has to travel, `(- N pos)`. The auxiliary function `optimal-jump-speed` solves this quadratic equation and returns the best-estimate, '(ceiling largest-positive-root), where largest-postie-root is the largest positive root of the quadratic equation.

We want our agent to prioritize jump-states that allow for it to travel the maximal distance within the given number of steps. The jump-state from which it is impossible to slow down so as to reach the goal state with a momentum of 1,`(> m (add1 largest-positve-integer))`, should be strongly discouraged; `jump-heursitic` returns `+inf.0` for such jump-states. For jump-states with momentum equal to or 1 more than the best-estimate, the function returns the best-estimate. For everything else, the function returns `(add1 best-estimate)`, so as to incentivize the jump-states that return `best-estimate` without completely demoting it in the pecking order.

> The choice of adding 1 is, for the most parts, arbitrary. There is a possibility that adding a different number would lead to a fewer number of node expansions for most of the jump-problems.

**Part B**

`jump-heuristic` is **not** admissible. Notice that for each jump-state, `optimal-jump-speed` returns the fewest number of steps required to reach goal state from its position, which is locally bound as best-estimate in `jump-heuristic`. `jump-heuristic` returns `(add1 best-estimate)` even for jump-state with momentum equal to `(- best-estimate 1)`. This heuristic ignores the fact that the agent can speed up by 1 or 2 while taking its next step. Since there are jump-states for which `jump-heuristic` overestimates the cost (number of steps), it is not admissible.

**Additional Notes**

`jump-heuristic` could be modified to be an admissible heuristic by making it return a value of best-estimate for jump-states with (best-estimate - 2) $\leq$ momentum $\leq$ (best_estimate + 1). We kept the current version of `jump-heuristic` because this implementation expands considerably fewer nodes. We posit this to be the case because our current implementation of `jump-heuristic` ignores potentially optimal paths, resulting in a fewer node expansions. The current implementation of jump-heuristic works pretty well both in terms of finding the optimal solution and in terms of efficiency. So, we decided against making our heuristic admissible.

**Problem 3**

| | Search Cost (nodes generated) | | | Effective Branching Factor | | |
|---|---|---|---|---|---|---|
| *depth* | **IDS** | **A*(*h1*)** | **A*(*h2*)** | **IDS** | **A*(*h1*)** | **A*(*h2*)** |
| 2 | 4 | 2 | 2 | 1.56 | 1.00 | 1.00 |
| 3 | 11 | 3 | 4 | 1.81 | 1.00 | 1.00 |
| 5 | 63 | 5 | 7 | 2.01 | 1.00 | 1.11 |
| 7 | 671 | 10 | 9 | 2.34 | 1.08 | 1.06 |
| 9 | 12683 | 11 | 22 | 2.71 | 1.04 | 1.175 |
| 12 | 725074 | 21 | 218 | 2.98 | 1.08 | 1.41 |
| 15 | too long | 51 | 498 | - | 1.14 | 1.39 |
| 17 | - | 73 | 2036 | - | 1.15 | 1.46 |
| 24 | - | 242 | 69681 | - | 1.16 | 1.52 |

**Extra Credits**

The second heuristic we implemented is a simple conditional that prefers speeding up when the agent is less than halfway to the goal, and prefers slowing down when the agent is more than halfway to the goal. This simple conditional makes it perform much better than thoughtless IDS but is still vastly outperformed by our original heuristic as the depth of the solution increases. The original heuristic ignores any impossible state where it doesn't have enough time to slow down for the goal, whereas this new heuristic does not. An optimal solution to the jump problem probably always needs to start slowing down before the halfway point, so this new heuristic will consider states that don't lead to any solution.