Angel He

Experimentation

Table of Contents

Code Overview

All changes made to the provided Code Base are shown in Table 1.0.

Table 1.0: Modules/files that were modified for the implementation.

Module/file	Changes
`search`	• Implemented the function `game_dijkstra_search`. • Added the optimization function `link_straight_flows`, with a helper function `path_blocked`.
`engine`	• Added the function `game_pos_is_free`, which functions the same as `game_is_free` but checks free positions using `pos_t` instead of coordinates. • Line 43 in `engine.c`: Modified the function `game_can_move`, by replacing the valid-color assertion with an if-statement. • Line 204 in `engine.c`: Removed the valid-color assertion in the function `game_next_move_color`, so that it returns -1 when there are no colors left to be picked.
`extensions`	Implemented the dead-end detection function `game_check_deadends`, with a helper `function is_deadend_cell`.
`utils`	• Added the function `pos_valid` which functions the same as `coords_valid` but checks valid positions using `pos_t` instead of coordinates. • Added the function `color_completed`, testing whether the goal cell of that color is reached.
`options`	Added an option for search with optimization (`"-O, --optimizations solve with straight-flow optimization"`).
`flow_solver.c`	Modified the display of search time from 3 to 7 decimal places.

Dead-end Detection Algorithm

The dead-end detection algorithm is based on the idea of checking whether a cell has a neighbour that allows a path entering it and/or leaving it. Table 2.0 summarises the conditions for which a cell would not be considered a dead-end, as implemented in the function `is_deadend_cell`. Note that an actual free neighbor (i.e. TYPE_FREE) is a wildcard that can either act as an entrance (`can_enter_from`) or an exit (`can_go_to`) for the cell being checked. Also, for different types of cells, the definition of an entrance or an exit cell may be different.

Besides, an exemption condition for all cells regardless of their type is if they have at least two actual free neighbors (checked with the provided `game_num_free_coords` function). In that case, they are immediately classified as a non-dead-end cell which requires no further checks with the function `is_deadend_cell`. A path body cell (TYPE_PATH but is not the head of a flow) also does not count as a dead-end.

Table 2.0: Conditions required for each type of cell to be classified as a non-dead-end.

Cell type	Condition	Valid entrance cell type	Valid exit cell type
free (TYPE_FREE)	Has a valid entrance AND exit	• a path head • a free cell • an initial cell whose flow has not yet started	• an unreached goal cell • a free cell
initial (TYPE_INIT)	Has a valid exit		• a path cell of the same color • a free cell
goal (TYPE_GOAL)	Has a valid entrance	• a path head of the same color (i.e. color completed) • a free cell	
path head (TYPE_PATH but is the head of a flow)	Has a valid exit		• a goal cell of the same color • a free cell

Idea for Optimization

For simplicity, the latter sections will use the following variables and short-hands:

- n = initial number of free cells
- t = execution time in seconds
- N = number of nodes generated
- c = number of colors
- w = width of the (square) puzzle board = $\sqrt{n+2c}$ ($n+2c$ = total number of cells)
- r = regular puzzle
- j = jumbo puzzle
- e = extreme puzzle

An extra optimization implemented for the flow-free solver is to detect linkable "straight flows" in the initial state. Examples of such "straight flows" are shown in Figure 1.0. Note that the `link_straight_flow` function essentially develops the original root as far as possible (like building a linked list from the original root) and returns a new root which possibly contains a few already completed paths, from which the Dijkstra search can then run.

The optimization stands on a simple heuristic: Since the aim of the game is to connect each initial cell to the goal cell of the same color, then in most cases, it would be desirable to minimize the crossings between different-colored paths. Therefore, if there is a pair of matching initial cell and goal cell lying on the same row/column, why choose a more complicated path than directly connecting them? A puzzle with all paths arranged row-by-row or column-by-column would be the best case, such that the Dijkstra search does not even need to happen in order to find the solution.

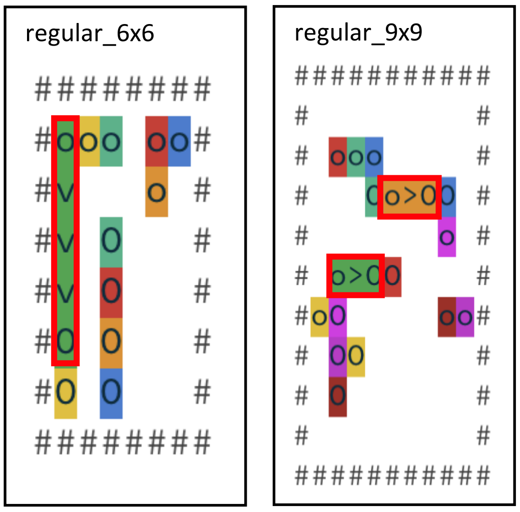


Figure 1.0: Examples of linked straight flows (bounded in red).

However, this optimization has two drawbacks. Primarily, this optimization is not complete, especially for complex puzzles. Consider the case illustrated in Figure 1.1 – the green straight flow would be detected and connected at the beginning. Yet, this would block out the red initial cell's path, which must cross that column occupied by the green flow in order to reach its goal cell. This would thus render the puzzle as unsolvable.

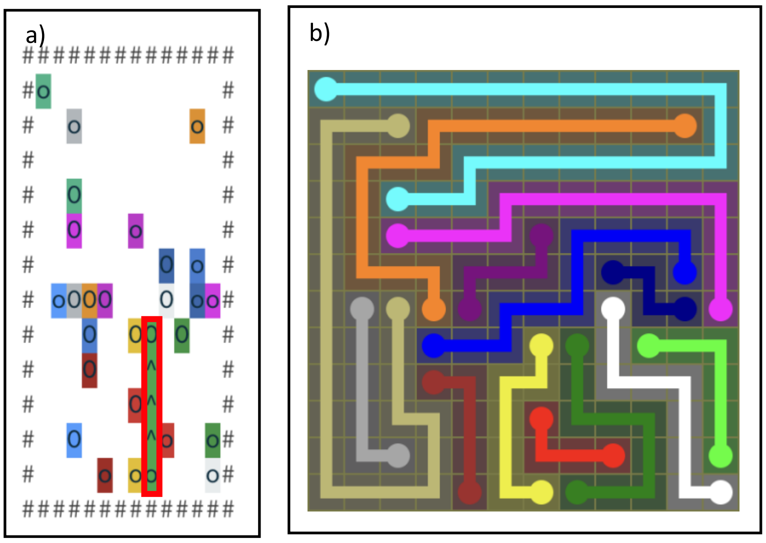


Figure 1.1: Connecting the vertical green straight flow (bounded in red) makes the puzzle unsolvable.
a) State of the jumbo 12x12 puzzle after connecting all linkable straight flows;
b) Solution to this puzzle (Zucker, 2016).

Secondly, this is not a constant-time optimization. The `link_straight_flow` function:

1. Checks through all pairs of initial and goal cells → $O(c)$
    - For each pair found in the same row/column, runs the `path_blocked` function to see if the straight flow is linkable → $O(w)$;
2. Loops through all colors again → $O(c)$
    - For each valid straight flow, creates the child nodes to connect the flow → $O(w)$.

The `path_blocked` function checks through all positions between the initial and goal cell. There are a maximum of $w - 2$ positions between them (i.e. when the initial and goal cell each occupies a corner), therefore the time complexity of this function is $O(w)$.

Hence, the overall time complexity of this optimization is $O(cw)$, where $w = \sqrt{n + 2c}$. For a typical puzzle board where $w \approx c$, this complexity becomes $O(w^2) = O(n + 2c) = O(n)$ since $2c \ll n$.

Preliminary Analysis

Consider the state space graph with:

- branching factor 4 (each direction is a branch),
- depth of the goal n (if it is mandatory to occupy all n free spaces),
- maximum total number of nodes m.

The number of nodes generated (N) would be $O(\min(4^n, m))$, as the state space is explored in "level order". The space complexity of the flow-solving algorithm would be the same as its time complexity since all generated nodes need to be stored in memory. Therefore, N and t are expected to grow exponentially with n.

However, practically, their growth rate is not expected to strictly adhere to this expression. Apart from the discrepancy between the underlying assumptions of the RAM Model of Computation and how computers actually perform (which especially influences t), there are also other factors affecting N and t apart from n.

A key factor is the puzzle's configuration, which affects not only the optimization and dead-end detection, but also the color choice. The default branch ordering uses the "most constrained" approach, where the solver continues with the last color moved if possible, otherwise picks the incomplete color with the fewest free spaces. Therefore, the number of free cells around each color's initial cell would affect the first color being picked, which in turn influences which states would be created before the solution path is found.

Experimentation Results

Note: t varies for each execution, so the values recorded were the average of 5 trials. The highlighted cells are some ‘anomalies’, which are explained in the “Further Discussion” section.

Table 3.0: The execution time and number of nodes generated by the flow-solver in different modes.

puzzle	n	Flow-solver mode							
		Basic		**-d**		**-O**		**-d -O**	
		t	N	t	N	t	N	t	N
r_5x5	15	0.000	18	0.000	15	0.000	16	0.000	13
r_6x6	24	0.000	283	0.000	34	0.000	143	0.000	29
r_7x7	37	0.002	3,317	0.001	157	0.002	3,317	0.001	157
r_8x8	52	0.326	409,726	0.005	712	0.314	409,726	0.006	712
r_9x9	63	0.446	587,332	0.014	1,514	0.371	341,388	0.011	1,161
j_10x10	76	0.252	348,980	0.063	6,288	0.139	198,020	0.042	4,212
j_11x11	97			3.227	283,887			1.280	111,690
e_8x8_01	54			0.023	3,019			0.023	3,019
e_9x9_01	73			0.526	78,523			0.535	78,523
e_9x9_30	69			0.261	30,100			0.235	30,100
e_10x10_01	90			1.440	174,533			1.447	174,533
e_10x10_30	88			2.535	300,524			2.548	300,524

Table 3.1: Comparison of the execution time and number of nodes generated by the basic solver, using different modes of branch ordering, over regular puzzles.

Puzzle	Branch ordering mode			
	Constrained (default)		**Randomized (-r)**	
	t	N	t	N
r_5x5	0.000	18	0.000	18
r_6x6	0.000	283	0.000	432
r_7x7	0.002	3,317	0.002	2,592
r_8x8	0.326	409,726	0.079	111,110
r_9x9	0.446	587,332	0.410	498,401

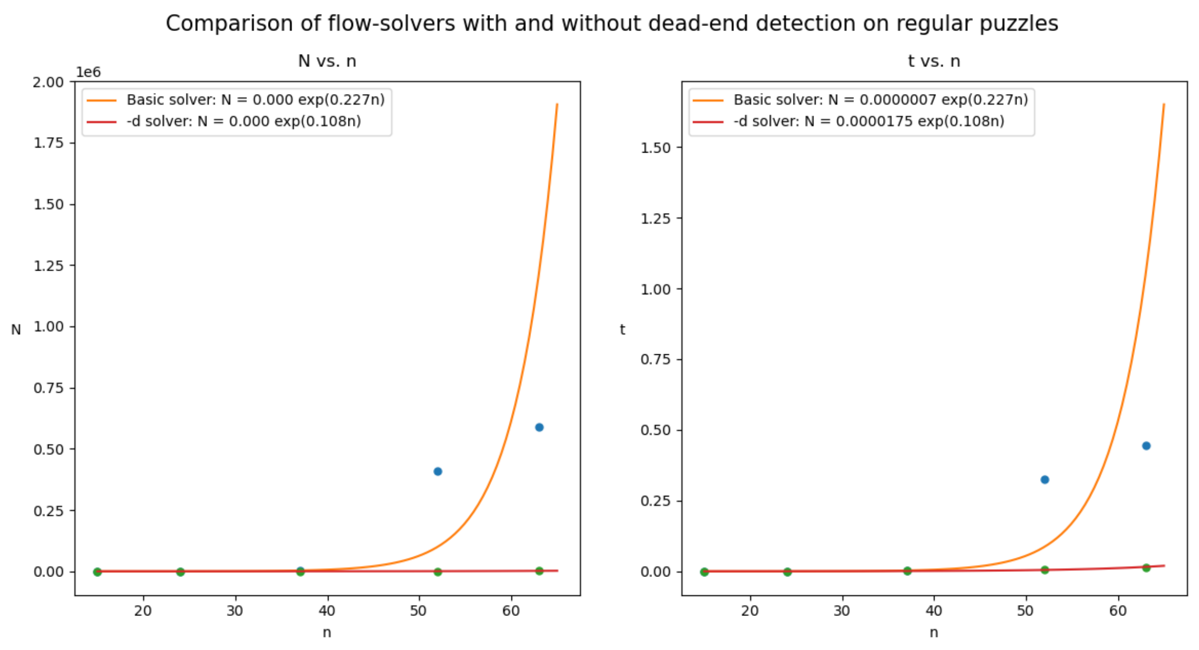


Figure 2.0: Comparison of the flow-solvers' performance with and without dead-end detection on regular puzzles, based on N and t.

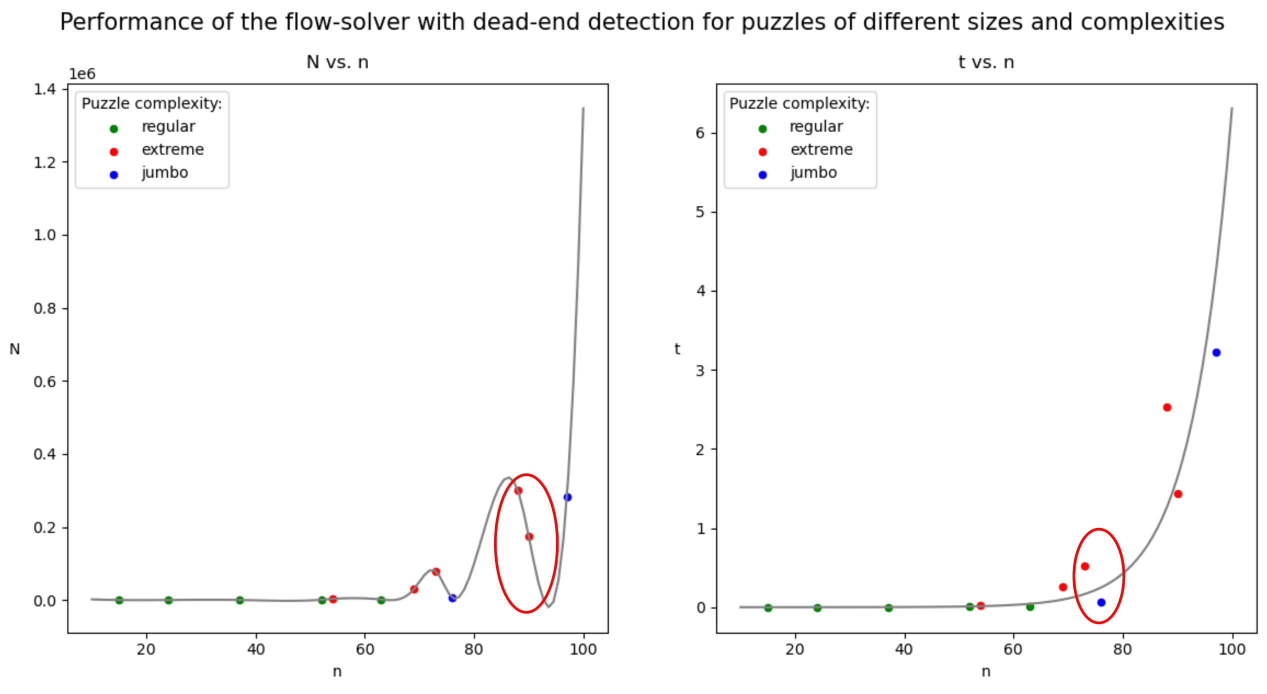


Figure 2.1: Performance of the -d flow-solver for puzzles of different sizes and complexities. The red circles indicate some positions where a higher n corresponded to a lower y-value.

Conclusion

Several key conclusions may be drawn from the experimentation:

1. Dead-end detection provides significant computational benefits. The solver with dead-end detection always performed at least as well as the basic solver, in terms of both N and t.
2. Both N and t grows approximately exponentially with n when performing search over the regular puzzles.
   But with dead-end detection, the growth rate is significantly reduced.
3. A larger n does not necessarily yield a larger N or t. t and N are not necessarily proportional to each other either.
   It is likely that t and N depends not only on n, but also on the puzzle's initial configuration and the branch ordering.
4. A randomized branch ordering may provide better results (lower t and N) than the constrained strategy on large puzzles.
5. Linking straight flows at the beginning can reduce N and t for puzzles that actually contain such straight flows.
   Yet, this optimization is not complete, and it potentially adds $O(n)$ time complexity to the flow-solving process.

For more detailed discussion explaining how these conclusions were drawn from the experimental results, please refer to the "Further Discussion" section.

Further Discussion

Point 1 & 2

As Figure 2.0 shows, the complexity of flow-solving over regular puzzles may be modelled as a power function of n.

For the solver with dead-end detection, the functions fitted were:

- $N = 3.494\ e^{0.099n} \approx O(\mathbf{1.10}^n)$
- $t = 0.0000175\ e^{0.108n} \approx O(\mathbf{1.11}^n)$.

For the basic solver, the functions fitted were:

- $N = 0.754\ e^{0.227n} \approx O(\mathbf{1.25}^n)$
- $t = 0.0000004\ e^{0.227n} \approx O(\mathbf{1.25}^n)$.

The power functions for the basic solver had a larger base, which matches its faster growth rate. Also, whilst the basic solver could only solve up to the jumbo 10x10 puzzle, the `-d` solver could solve all the way up to the extreme 10x10 puzzles, as evident from Table 3.0. Therefore, the dead-end detection mechanism clearly improves the flow-solver's performance, especially for high values of n.

Point 3

Notice that the power function was not strictly a perfect fit for the basic solver, and its N does not always increase with n, as highlighted in Table 3.0. Specifically, the basic solver generated fewer nodes to solve the jumbo 10x10 puzzle than the regular 9x9 puzzle; whilst the optimized solver (without dead-end detection) generated fewer nodes and required less time to solve the jumbo 10x10 puzzle, compared to the regular 8x8 and 9x9 puzzle.

As mentioned, this behaviour may imply that the complexity of the flow-solving algorithm is not entirely dependent on n. This idea is further endorsed by Figure 2.1, where N and t are not always increasing with n and varied with the different puzzle complexities/configurations. The dissimilar shapes of the graphs for N and t over n also indicated that N and t are not necessarily proportional to each other.

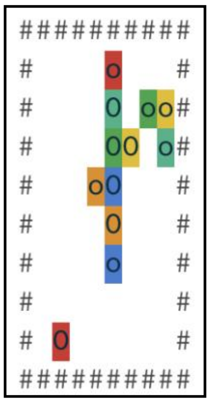
Point 4

An interesting observation was that t varied during each run. Particularly, the solver usually had a longer execution time on its first run than the latter runs, therefore the 5 trials used for Table 3.0 excluded the first run. This may be consequential to the C compiler's automatic code optimization, which attempts to improve the program run-time by making the intermediate code consume fewer resources (i.e. CPU, memory) (GeeksforGeeks, 2022).

Besides, Table 3.1 indicated that a randomized ordering often produces a lower N and t than the default constrained approach, especially for higher n-values. Compared to using the constrained ordering, the randomized solver generated much fewer nodes and took much less time to solve the regular 8x8 puzzle than the constrained solver. Yet, it generated more nodes for the smaller regular 6x6 puzzle. This suggests that picking the color non-deterministically may provide better results in solving an NP problem like flow-free, especially for large inputs.

Point 5

For the optimized solver, the degree of optimization really depends on how many linkable straight flows were present in the puzzle. This explains why the optimization did not reduce N at all for the regular 8x8 puzzle (shown in Figure 3.0), which has no linkable straight flows present.



```
##########
#    o   #
#    o oo#
#    oo o#
#   oo   #
#    o   #
#    o   #
#        #
#  o     #
##########
```

Figure 3.0: The initial configuration of the regular 8x8 puzzle which contains no linkable straight flows.

Appendix

The following figures are transformed from Table 3.0.

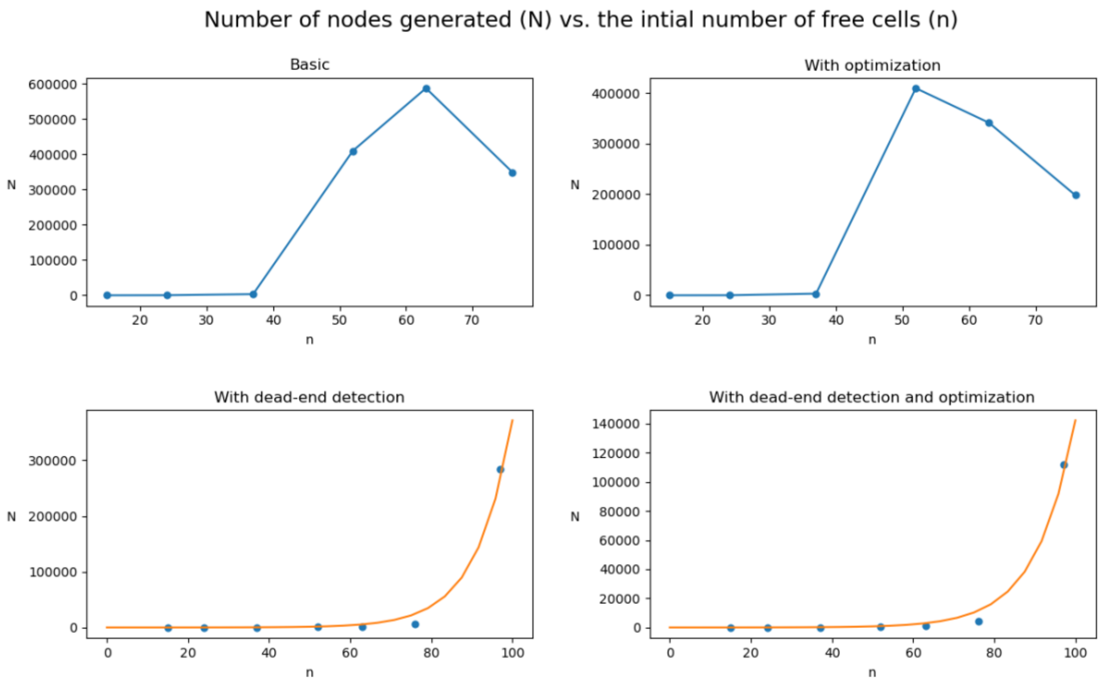


Figure 4.0: N vs. n for the flow-solvers of different modes.

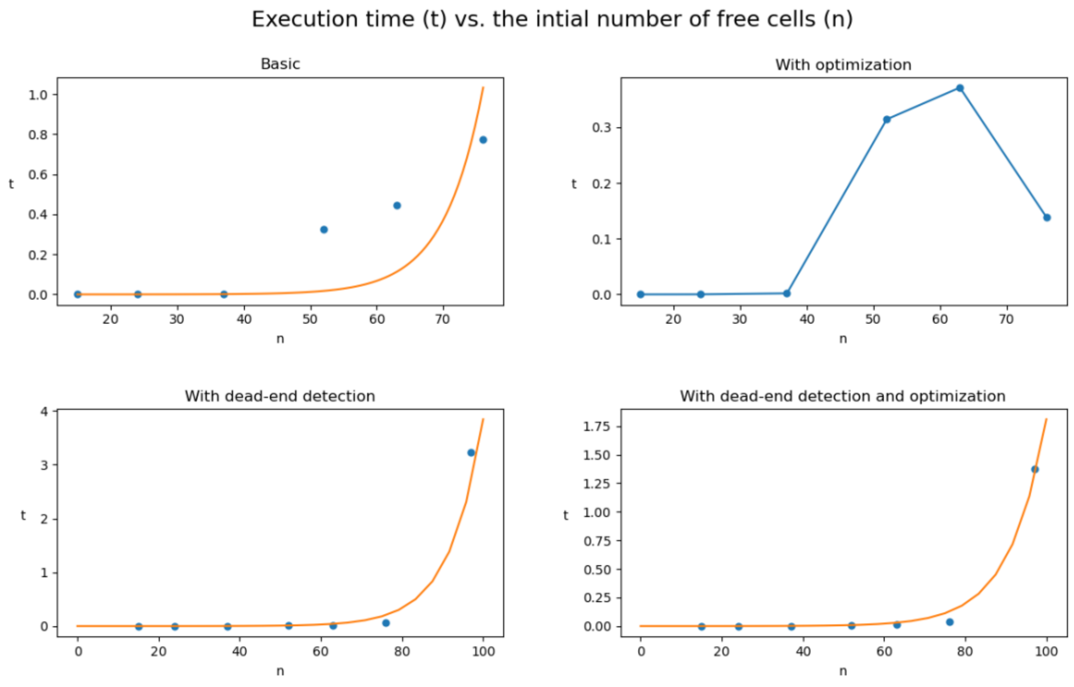


Figure 4.1: t vs. n for the flow-solvers of different modes.

References

GeeksforGeeks. (2022, June 23). *Code Optimization in Compiler Design*. https://www.geeksforgeeks.org/code-optimization-in-compiler-design/#:~:text=The%20code%20optimization%20in%20the,running%20machine%20code%20will%20result.

Zucker, M. (2016, August 28). *Flow Free Solver*. Needlessly Complex. https://mzucker.github.io/2016/08/28/flow-solver.html