Artificial Intelligence

Programming Project Part 1: Searchclient

Lucian Leahu

DIS Copenhagen Fall

4 October 2019

Team Name: Sejelo

Contributors: Sejal Dua & Angelo Williams

Group Declaration of Work Breakdown

IDEAS: Sejal and Angelo
PROGRAMMING: Sejal and Angelo
THEORETICAL: Sejal and Angelo
REPORT: Sejal and Angelo

———————————————————————

God of Debugging: Angelo
Aesthetíque: Sejal

———————————————————————

Special Features:
» bash scripting to automate benchmarking
» Excel table to pandas dataframe
» matplotlib visualizations

Project Overview

The project is partly inspired by the developments in mobile robots for hospital use and systems of warehouse robots like the KIVA robots at Amazon. In both applications, there is a high number of transportation tasks to be carried out.

Among the most successful and widely used implementation of hospital robots so far are the TUG robots by the company Aethon. TUG robots were first employed in a hospital in 2004, and is now in use in more than 100 hospitals in the US. Since 2012, TUG robots have also been applied at a Danish hospital, Sygehus Sønderjylland, the first hospital in Europe to employ them.

The goal of this programming project is to implement a simplified simulation of transportation robots at a hospital or in a warehouse.

Relevant topics discussed and implemented in this project include:

- Search Strategies
  - ⋆ Breadth First Search
  - ⋆ Depth First Search
- Optimizing
- State Spaces
- Benchmarking

Exercise 1: Search Strategies

Part A: Running the BFS client

The client contains an implementation of breadth-first search via the `FrontierBFS` class. Run the BFS client on the `SAD1.lvl` level and report your benchmarks.

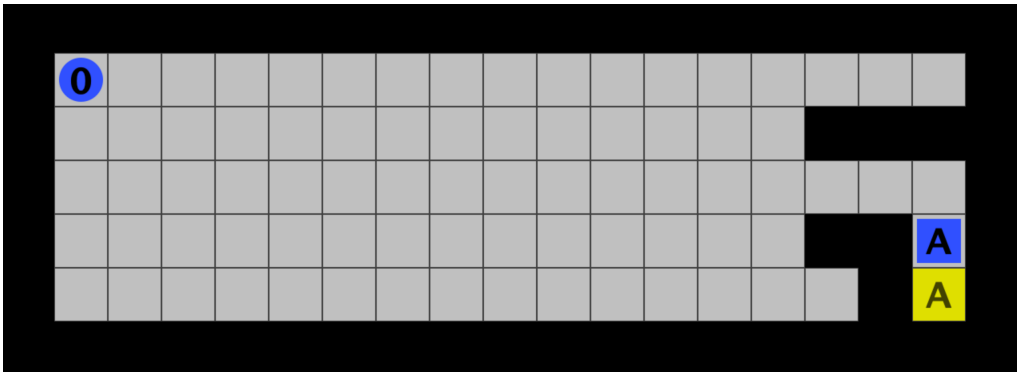


Figure 1: SAD1 level

Level	Frontier	Time (s)	Memory Used (MB)	Solution Length	States Generated
SAD1	BFS	0.151	25.28	19	80

Part B: Comparison of levels based on benchmarks

Run the BFS client on `SAD2.lvl` and report your benchmarks. Explain which factors make `SAD2.lvl` much harder to solve using BFS than `SAD1.lvl`. (You can also try to experiment with levels of intermediate complexity between `SAD1.lvl` and `SAD2.lvl`).

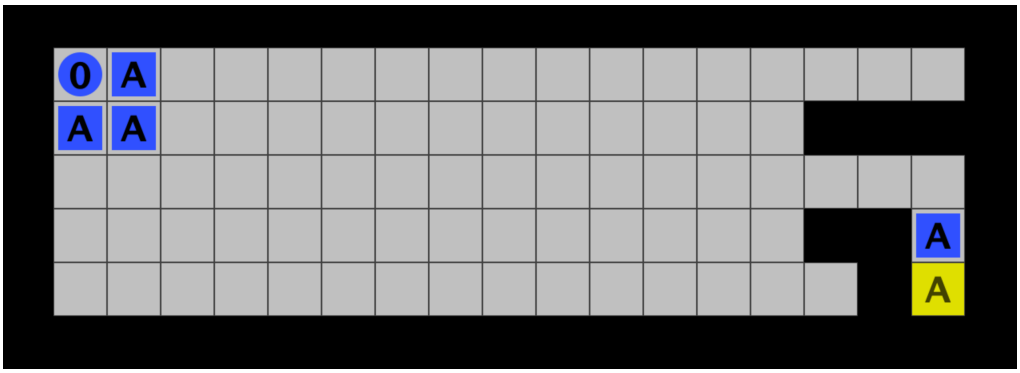


Figure 2: SAD2 level

Level	Frontier	Time (s)	Memory Used (MB)	Solution Length	States Generated
SAD2	BFS	67.547	8192	Not found	18195

`SAD1.lvl` is a fairly simple level. With no boxes present, the only possible actions for the agent is to move in any of the four directions. Therefore, the search space is much smaller. In `SAD2.lvl`, the frontier is much larger because there are three different boxes blockading the agent. The agent must push through these boxes in order to navigate to the opposite corner of the level (where the goal cell is) and push the most convenient box into the goal cell. These obstructions force the agent to generate a ridiculous amount of states because since it is BFS, all the moves for each box must be considered before the agent can continue paving its path to the yellow A.

Part C: Modifying implementation to support DFS

Modify the implementation so that it supports depth-first search (DFS). Specifically, implement the class `FrontierDFS` such that `SearchClient.search()` behaves as a depth- first search when it is passed an instance of this frontier. Benchmark your DFS client on `SAD1.lvl` and `SAD2.lvl` and report the results.

Modified Implementation:

```
class FrontierDFS
    implements Frontier
{

    // emulated Stack operations for DFS by implementing LIFO invariant on
    // Java ArrayDeque data structure
    private final ArrayDeque<State> stack = new ArrayDeque<>(65536);
    private final HashSet<State> set = new HashSet<>(65536);

```

```
    @Override
    public void add(State state)
    {
        // everything is the same here
        this.stack.addLast(state);
        this.set.add(state);
    }

    @Override
    public State pop()
    {
        // call pollLast (pop) rather than pollFirst here
        State state = this.stack.pollLast();
        this.set.remove(state);
        return state;
    }

    // nothing out of the ordinary below... use FrontierBFS class as template
    //
```

As you can see in the comments, the implementation was modified to support DFS by way of the class **FrontierDFS** which **implements Frontier**. As opposed to the **FrontierBFS** class, this class uses an **ArrayDeque** Java data structure which maintains the LIFO (last in, first out) invariant to behave like a stack. Instead of adding to the back of the data structure and popping from the front, States are added to the back and popped from the back. The rest of the implementation is identical to that of the **FrontierBFS** class.

Level	Frontier	Time (s)	Memory Used (MB)	Solution Length	States Generated
SAD1	DFS	0.102	15.02	27	75
SAD2	DFS	0.089	12.39	25	86

The table above suggests that the DFS client implemented in Part C performs far better on **SAD1.lvl** and **SAD2.lvl** than the BFS client does. First of all, it is able to actually find a solution for **SAD2.lvl** without exceeding the allocated memory (8 GB) for the task. Moreover, it solves both levels fast, with minimal memory usage, and minimal states generated because DFS is about pursuing paths to their deepest possible states.

Part D: Explaining difference between number of generated states (BFS)

On the levels **SAD1.lvl** and **SAD2.lvl**, DFS is much more efficient than BFS. But this is not always the case. Run BFS and DFS on the level **SAfriendofBFS.lvl** and report how many states are generated by each of the two algorithms. Why is there such a huge difference between the number of generated states? For this question, you need to fill in the relevant lines of Table 2 as well as give a brief, but conceptually precise, explanation of why there is such a big difference in the performance of the two algorithms on this level.

Level	Frontier	Time (s)	Memory (MB)	Solution Length	States Generated
SAfriendofBFS	BFS	0.455	135.57	3	1227
SAfriendofBFS	DFS	24.14	8192	Not found	30012

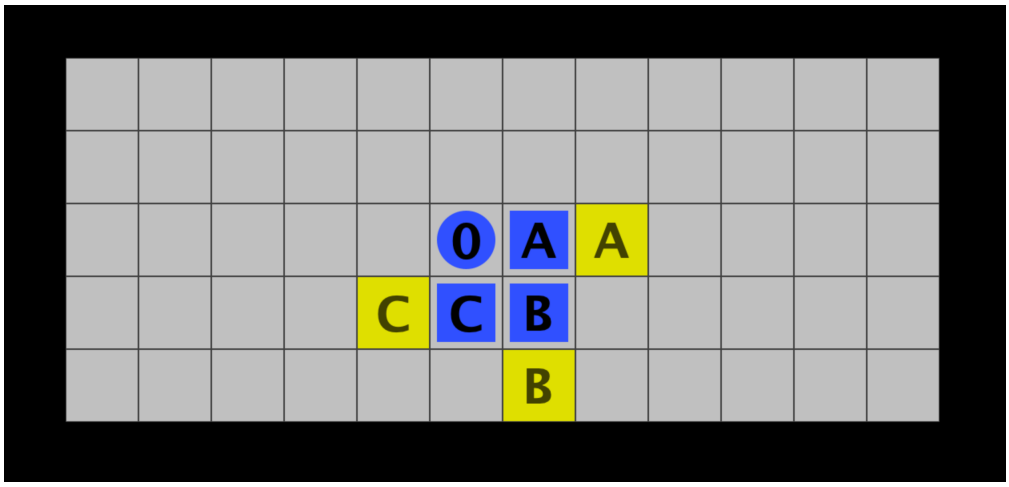


Figure 3: SAfriendofBFS level

BFS generates states one breadth level at a time from the starting state. For this level, there is a solution that requires relatively few moves, so the BFS client only needs to generate the states in the inner ring in close vicinity from the initial state. Thus, the amount of states the BFS client must generate in order to find the solution of length 3 is only 1227– it seems like it is a lot, but it is actually a relatively low amount. On the other hand, DFS generates a deep sequence of states with no concern to proximity. The DFS agent is incapable of finding the 3 move solution because it instead must wander around the level, pursuing each path to a dead end state, until, by chance, the three boxes reach their spots. As a result, it ends up generating 30012 states until exceeding memory usage.

Part E: Explaining difference between number of generated states (DFS)

Run BFS and DFS on the level `SAfriendofDFS.lvl` and report how many states are generated by each of the two algorithms. Why is there such a huge difference between the number of generated states? For this question, you need to fill in the relevant lines of Table 2 as well as give a brief, but conceptually precise, explanation of why there is such a big difference in the performance of the two algorithms on this level.

Level	Frontier	Time (s)	Memory (MB)	Solution Length	States Generated
SAfriendofDFS	BFS	58.089	8122.34	8	30799
SAfriendofDFS	DFS	0.171	36.94	60	305

There also exists a solution with relatively few moves for this level, however there are many possible actions available to the agent due to the amount of boxes. Because the level is so populated, the BFS state space grows incredibly fast. Luckily for the DFS client, it is very easy to stumble into a solution by making sequential actions and barging through a deep path in such a way that boxes naturally get shoved into goal cells. This means that the DFS client can solve the level in a fraction of a second, generating only 305 states in order to do so. It is worth noting here that because we allocated 8 GB for the BFS frontier on `SAfriendofDFS.lvl`, it happened to find the solution just before exceeding memory usage, but it still generated

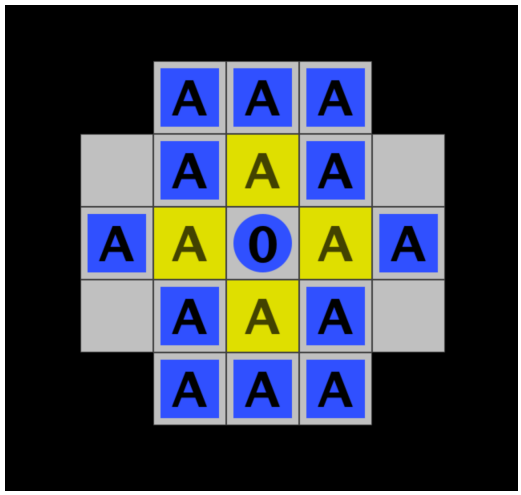


Figure 4: SAfriendofDFS level

30799 states, so it was by no means efficient.

Part F: SAFirefly and SACrunch

Benchmark the performance of both BFS and DFS on the two levels `SAFirefly.lvl` and `SACrunch.lvl`, shown in Figure 4. For this question, you only have to fill in the relevant lines of Table 2.

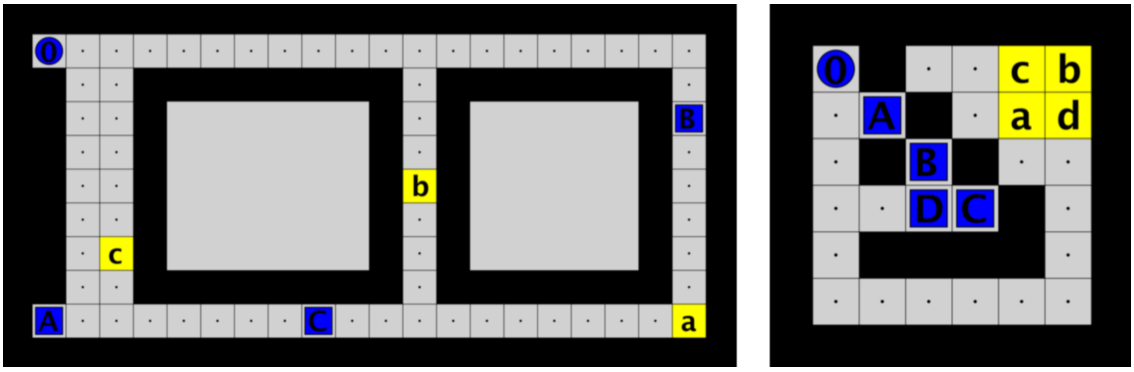


Figure 5: SAFirefly and SACrunch levels

Clearly, our BFS and DFS implementations are nowhere near optimized enough to solve `SAFirefly.lvl` and/or `SACrunch.lvl` at this point. It is also interesting to note that even after being given 8 GB of memory to work with, the respective clients use it all up in around 30 seconds or less, so the current algorithmic process of generating states blows through memory at a staggering state. This must be fixed if we are to successfully solve these two levels.

Level	Frontier	Time (s)	Memory (MB)	Solution Length	States Generated
SAFirefly	BFS	27.82	8192	Not found	81629
SAFirefly	DFS	24.06	8192	Not found	87730
SACrunch	BFS	34.701	8192	Not found	87572
SACrunch	DFS	19.647	8192	Not found	73865

Complete table of benchmarks

Table 1: Benchmarks - Exercise 1					
Level	Frontier	Time (s)	Memory Used (MB)	Solution length	States Generated
SAD1	BFS	0.151	25.28	19	80
SAD1	DFS	0.102	15.02	27	75
SAD2	BFS	67.547	8192	N/A	18195
SAD2	DFS	0.089	12.39	25	86
SAfriendofDFS	BFS	58.089	8122.34	8	30799
SAfriendofDFS	DFS	0.171	36.94	60	305
SAfriendofBFS	BFS	0.455	135.57	3	1227
SAfriendofBFS	DFS	24.14	8192	N/A	30012
SAFirefly	BFS	27.82	8192	N/A	81629
SAFirefly	DFS	24.06	8192	N/A	87730
SACrunch	BFS	34.701	8192	N/A	87572
SACrunch	DFS	19.647	8192	N/A	73865

Exercise 2: Optimizations

Optimization 1

Flaw: the location of walls and goal cells are static (i.e. never changes between two states), yet each state contains its own copy.

Improvement in the code: the data structures representing walls and goals in the State class were changed from member variables to class variables. This means that every state for a level will reference just one static class variable for the walls 2D array and one for the goals 2D array. The most significant modification is provided below:

```
public static boolean[][] walls;
public static char[][] goals;
```

Significance of improvements: As seen in Table 2 below, in most levels the memory used was at least halved and the time spent searching states was greatly decreased.

Table 2: Optimized Implementation #1 - Exercise 2					
Level	Frontier	Time (s)	Memory Used (MB)	Solution length	States Generated
SAD1	BFS	0.087	15.68	19	80
SAD1	DFS	0.13	11.4	27	75
SAD2	BFS	67.189	8192	N/A	219020
SAD2	DFS	0.069	9.12	25	86
SAfriendofDFS	BFS	17.149	3307.72	8	89112
SAfriendofDFS	DFS	0.101	15.34	60	305
SAfriendofBFS	BFS	0.288	60.2	3	1227
SAfriendofBFS	DFS	28.762	8192	N/A	204410
SAFirefly	BFS	43.665	8192	N/A	221864
SAFirefly	DFS	34.056	8192	N/A	219263
SACrunch	BFS	48.133	8192	N/A	225596
SACrunch	DFS	31.986	8192	N/A	213472

Optimization 2

Flaw: when we construct the initial state, the widths and heights of the walls and boxes arrays are set to 130 regardless of the actual size of a level, and the number of colors are similarly fixed.

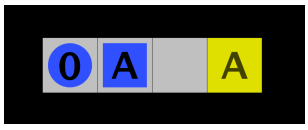
Improvement in the code: we generate the arrays in the same fashion as before, but instead of using a regular array, we use a dynamic array that is implemented via an `ArrayList`. This ensures the dimensions are only as big as needed. The cost of traversing the level once in order to obtain the correct sizing of the level is not too expensive because once we get the initial state, every subsequent state will be initialized with these dimensions, saving us a ton of excess memory that would have otherwise gone to waste. After generating the dynamic array representation of the level, we copy the contents over to the standard array used in the original code in order to ensure compatibility with the rest of the code base. To resolve the issue with the fixed colors, we also implemented an `ArrayList` to add colors as needed. This modification affected some other parts of `State.java`, but we believe it made a noticeable difference in performance benchmarks.

Significance of improvements: As seen in Table 3 below, in most levels the memory used was at least halved and the time spent generating redundant states was greatly decreased. In other words, the impact of the optimizations was very significant. The BFS and DFS clients are now able to solve all levels except for `SACrunch.lvl` for the BFS client. This is a huge improvement from before, and none of the levels terminate due to the 8 GB of memory being exceeded.

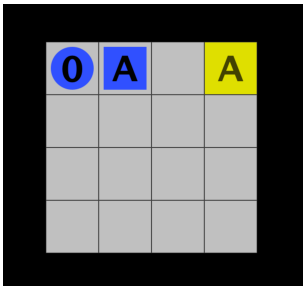
Table 3: Optimized Implementation #2 - Exercise 2					
Level	Frontier	Time (s)	Memory Used (MB)	Solution length	States Generated
SAD1	BFS	0.029	3.72	19	80
SAD1	DFS	0.025	3.59	27	75
SAD2	BFS	14.467	692.72	19	635190
SAD2	DFS	0.022	3.59	25	86
SAfriendofDFS	BFS	1.185	72.13	8	89112
SAfriendofDFS	DFS	0.031	3.72	60	305
SAfriendofBFS	BFS	0.048	5.12	3	1227
SAfriendofBFS	DFS	35.552	2015.72	981528	2953986
SAFirefly	BFS	18.261	2252.72	60	1961416
SAFirefly	DFS	24.539	4101.72	2517074	4089953
SACrunch	BFS	179.986	3840.72	N/A	6464388
SACrunch	DFS	9.218	4392.28	380992	1023377

Exercise 3: State Space Growth

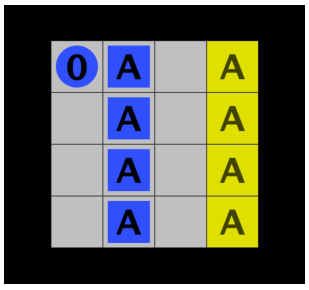
The level `SAsoko1_08.lvl` has size 8. What are the largest levels of each type that your BFS client can currently handle? Why is there such a big difference between the biggest size level you can handle of the three types? To complete this exercise, you should: 1) Include benchmarks of the biggest sizes of `SAsoko1`, `SAsoko2` and `SAsoko3` levels that your BFS client can solve 2) Provide a brief explanation of why there is such a big difference between the biggest size level you can handle of the three types. Try to make your answer as mathematically precise as possible. Hint: think about the state space sizes.



(a) SAsoko1



(b) SAsoko2



(c) SAsoko3

Figure 6: Comparison of SAsoko types for size 4

Biggest size BFS client can handle for `SAsoko1`: 128
Biggest size BFS client can handle for `SAsoko2`: 32
Biggest size BFS client can handle for `SAsoko3`: 5

Table 4: State Space Growth Experiment - Exercise 3					
Level	Size	Time (s)	Memory Used (MB)	Solution length	States Generated
SAsoko1	4	0.016	3.59	2	3
SAsoko1	8	0.017	3.59	6	15
SAsoko1	16	0.024	3.59	14	68
SAsoko1	32	0.034	4	30	266
SAsoko1	64	0.054	6.28	62	1034
SAsoko1	128	0.15	10.09	126	4128
SAsoko2	4	0.019	3.59	2	44
SAsoko2	8	0.047	5.4	6	660
SAsoko2	16	0.329	21.27	14	8580
SAsoko2	32	3.686	515.72	30	118093
SAsoko2	64	98.825	8192	N/A	808491
SAsoko2	128	65.836	8192	N/A	221993
SAsoko3	4	0.429	11.63	8	17016
SAsoko3	5	60.161	820.72	15	874805
SAsoko3	6	178.28	1900.72	N/A	3287110
SAsoko3	7	178.01	2834.72	N/A	4032039
SAsoko3	8	179.13	3482.72	N/A	4734372
SAsoko3	16	178.7	6819.72	N/A	5173127
SAsoko3	32	93.528	8192	N/A	2506572
SAsoko3	64	57.46	8192	N/A	810381
SAsoko3	128	35.397	8192	N/A	224619

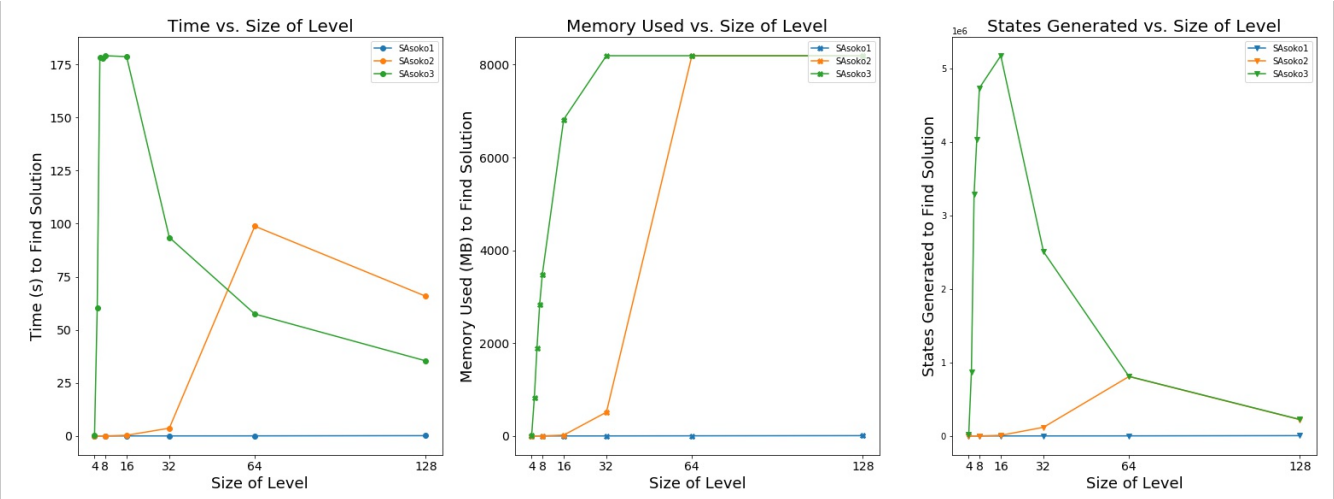


Figure 7: Visualization of various metrics vs. size of each class of SAsoko level

There is such a big difference in the biggest size level the BFS client can handle for each type of `SAsoko` level because as you can see in Figure 6, the dimensionality of the initial state is different for each type of `SAsoko` level. In `SAsoko1`, we can see that we have a 1D hall-way type layout. The BFS client performs well on this because there is no choice but for it to push the

box forward until it reaches the goal cell. In fact, the statistics report is so intuitive that we can even notice that the solution length is equal to the size - 2. The number of states generated gets multiplied by around 4 from one size to the next. The BFS client can handle all sizes for this type. For `SAsoko2`, we can refer to the states as 2 dimensional with a 1 dimensional solution. The BFS client has to generate significantly more states for each size, but the solution ends up being a linear path. Interestingly, the number of states generated gets multiplied by around 16 from one size to the next. By the time the size increases to 64, there are simply too many states generated to stay under 8 GB of memory usage. The BFS client can only handle size 32 for this type. For `SAsoko3`, there is a 2D state layout with a 2D solution and no limiting walls. The state space, once again, grows exponentially, but it grows beyond control here. Before we even get a memory exceeded message, we get a timeout on sizes 6, 7, 8, and 16. Then, as the dimensions of the board get bigger and bigger, the number of breadth levels that must be explored by the BFS client grow uncontrollably and the number of states needed to be generated also grow out of control. This causes the level to be terminated due to maximum memory usage instead of a timeout.

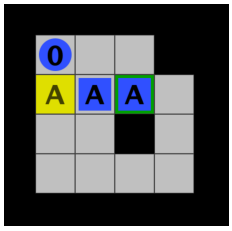
Exercise 4: Other Levels & Domains

Try to test your client on some of these levels, and see whether your optimised client can solve them (using BFS). Since Sokoban does not allow pulls and you can't push around corners, you will have to modify `Actions.java` for this exercise. You will also have to create the level files yourself for the levels you'd like to try out. Also try out your client on the labyrinth levels `SAlabyrinth.lvl`, `SAlabyrinthOfStBertin.lvl`, and `SAmicromousecontest2011.lvl` (the latter is the one from the micro mouse contest that was considered in one of the previous exercises). Why are these labyrinth levels so much easier to solve than most of the previously considered levels in this assignment? For this exercise, include the benchmarks of your selected Sokoban levels. Make sure to state the level numbers. Also include the discussion on why the labyrinth levels are simple for the search client (but no need to include the benchmarks for those).

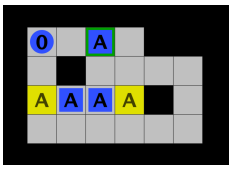
Figure 8: LabyrinthOfStBertin starting state for visual context

Table 5: Benchmarks for Labyrinths - Exercise 4					
Level	Frontier	Time (s)	Memory Used (MB)	Solution length	States Generated
SAlabyrinth	BFS	0.213	20.11	676	1753
SAlabyrinth	DFS	0.144	17.97	676	1082
SAlabyrinthOf...	BFS	0.17	10.66	1110	1150
SAlabyrinthOf...	DFS	0.122	10.78	1110	1150
Samicromouse...	BFS	0.13	9.12	112	535
Samicromouse...	DFS	0.081	7.4	154	396

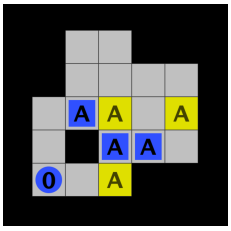
This exercise primarily has to do with how the BFS and DFS clients perform when we give it a level with many walls and fewer states to generate. The labyrinth levels are so much easier to solve than the previously considered levels for this assignment because the client can only move forward for around 90% of the time. Every once in a while, a turn must be made, but the labyrinth levels are almost so constricted that it does not matter whether the frontier is BFS or DFS: the walls limit the amount of potential states in such a significant way that there is little difference. Take a look at each pair of rows in the "States Generated" column in Table 5. We get comparable results for each set of labyrinth trials, irregardless of the frontier / client that is used.



(a) Sokoban Level 1



(b) Sokoban Level 35



(c) Sokoban Level 70

Figure 9: Starting state of Sokoban Levels 1, 35, and 70 for visual context

Table 6: Benchmarks for Sokoban Levels - Exercise 4					
Level	Frontier	Time (s)	Memory Used (MB)	Solution length	States Generated
SAsokoban01	BFS	0.051	5.12	34	749
SAsokoban02	BFS	0.068	7.68	23	1983
SAsokoban03	BFS	0.088	10.28	31	2960
SAsokoban04	BFS	0.253	21.74	25	12520
SAsokoban05	BFS	0.461	29.66	37	22718
SAsokoban35	BFS	0.166	10.35	53	8061
SAsokoban36	BFS	0.306	24.18	59	18041
SAsokoban37	BFS	0.731	29.24	53	40349
SAsokoban38	BFS	0.184	7.21	75	6358
SAsokoban39	BFS	0.174	12.54	60	7817
SAsokoban70	BFS	0.344	7.39	55	6050
SAsokoban71	BFS	0.147	6.35	36	5998
SAsokoban72	BFS	3.244	106.16	70	127852
SAsokoban73	BFS	1.88	91.88	70	133303
SAsokoban74	BFS	0.241	15.48	93	9762

We created a total of 15 Sokoban levels for our BFS client to try and tackle. We attempted to select each group of 5 at varying difficulty levels assuming that the actual Sokoban game has levels sorted in increasing order of difficulty. Our BFS client had no difficulties solving the levels, even with a modified `Actions.java` to only allow forward pushing. It executed almost every level with speed and minimal memory usage. It struggled the most with `SAsokoban70.lvl`. Try it yourself and see if you can outsmart our BFS client!