AIA: Evaluating the Effectiveness of Different Inference Engines using a Heist Scenario

James Healey
BSc Computer Science,
Teesside University,
m2019524@tees.ac.uk

George Milner
BSc Computer Science,
Teesside University,
m2049480@tees.ac.uk

Chris Percival
BSc Computer Science,
Teesside University,
e5116399@tees.ac.uk

Keywords: Clojure, NetLogo, inference, planner, operators, breadth search, legal move generator

Abstract

In this paper we compared and contrasted the efficiency and usability of the operator search and planner inference engines. We created operators for each inference engine in Clojure, then ran the engines using a heist scenario, and graphically represented the results using NetLogo. We found that although operator search allows for faster development, the planner was much more efficient in real-world scenarios.

1. Background

Inference engines apply logical rules to an existing knowledge base (world state) to infer new knowledge (Rajeev & Krishnamoorthy, 1996). The addition of the new knowledge in the world state may trigger more rules to be applied. This can be used to create planning mechanisms.

In this paper, we use the operator search and planner inference engines with operators to create a plan from a given world and start state to reach a specified goal state.

The operator search (ops-search) inference engine uses a legal move generator, forward chaining, and breadth-first search to achieve a given goal state (Lynch, ops-search, n.d.).

The planner engine uses means end analysis and STRIPS to build a goal stack and create a plan (Lynch, planner, n.d.), in a similar fashion to the General Problem Solver (GPS) (Newell, Shaw, & Simon, 1959).

2. Scenario

This paper aims to evaluate the usability and efficiency of the ops-search and planner inference engines using a bank heist scenario.

In the scenario, an agent (thief) will attempt to retrieve cash from the bank's vault, then navigate back out of the building to a getaway van. To access the vault, the agent will first need to acquire a key which can be placed in any location within the building.

The heist scenario was chosen as a scalable way to demonstrate a wide range of operators in a visually interesting way. This scalability was used when evaluating the performance of the different engines.

3. Introduction

Inference engines have many real-world applications, such as in expert systems. Ease of implementation and performance of these inference engines vary, and finding the best tools for the situation is very important when designing a system. More information on how these inference engines differ would allow developers to more easily pick the correct tool for their needs.

To evaluate, compare, and contrast the two chosen inference engines, we first created a world state, start state, and set of planner and ops-search operators in Clojure. The ops-search and planner inference engines were then used to create a path to a given goal state. Commands built into the operators were then sent via a socket to NetLogo where they are visualised. Tests were then run on the efficiency of the different inference engines, and the results presented.

4. Design – Issues and Decisions

When implementing the model for the scenario, there were a number of design decisions made. This section describes the problems encountered, possible solutions considered, and the reasoning behind the solution chosen.

4.1 World State Representation

The first decision that had to be made was how to represent the world state. Representing the world as a grid with coordinates (see Figure 1) could allow for more detailed world representations, but could also increase the pathfinding complexity and branching factor of the breadth-first search when using ops-search.

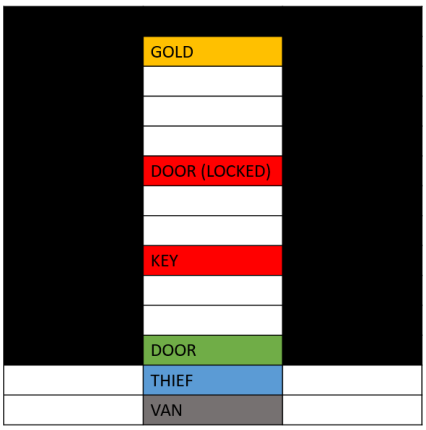

Figure 1 – Example grid representation

Figure 2 – Example abstract representation

As we only needed location-level representation in our scenario, we decided to represent the world state as a series of abstract locations (see Figure 2). These locations were represented as the following tuples:

```
(isa van location)
(isa lobby location)
(isa tellers location)
(isa storage location)
(isa vault location)
```

4.2 Door State Representation

Early in the design process, we encountered a problem relating to the representation of doors in the world state. There were two main solutions that were considered.

1 Doors referred to via their state – This approach only requires a single tuple for each direction, but would need both tuples to be changed when the door state changes.

```
(opened-door van lobby)
(opened-door lobby van)

(closed-door lobby tellers)
(closed-door tellers lobby)

(locked-door tellers vault)
(locked-door vault tellers)
```

2 Doors as explicit tuples – This approach allows multiple links between the same two locations but requires at least four tuples to define each link: one to define the door, two connect tuples to specify the associated locations, and one to define the door's current state.

```
(isa entrance-door door)
(connects van entrance-door)
(connects lobby entrance-door)
(state entrance-door closed)
```

The second option was chosen; although it requires more tuples in the world state, the changes required to the tuples when actions are performed are much simpler, as it only changes a single tuple. If the amount of tuples required in the world and start states became an issue, forward chaining could be used to add a default rule for doors being closed, which could then be overridden before the main loop with explicit statements for with different states.

5. Tuple Definitions

We modelled the scenario as a series of tuples that represent the locations and states of the items, agents, and locations. The type of tuples defined are described below.

Keyword	Description	Example
at	The location of an agent	(at thief van)
in	The location of an item	(in key tellers) (in cash vault)
isa	Specifies the type of something	(isa thief agent) (isa lobby location) (isa vault-door door)
connects	Connects a location to a door	(connects tellers vault-door)
state	Defines the current state of the door	(state entrance-door open) (state lobby-tellers-door closed) (state vault-door locked)
holds	The item that the agent is holding	(holds thief key) (holds thief empty)

Table 1 – Tuple definitions.

6. Inference engines

Clojure was chosen as the language to develop/consume the inference engines as it is built on Java, and allowed us to write lisp-like functional code whilst also communicating with external programs used to graphically display results. IntelliJ IDEA (JetBrains, 2016) with the La Clojure plugin (JetBrains, 2015) was used as the development environment.

6.1 Operator Search

The operator search inference engine is data-driven and uses breadth-first search and forward chaining to search for the required goal state. An example operator is shown below.

```
move {
:pre ((isa ?agent agent)
(at ?agent ?location)
(connects ?location ?door)
(connects ?location2 ?door)
(state ?door open))
:del ((at ?agent ?location))
:add ((at ?agent ?location2))
:cmd [move ?agent ?location2]
:txt (moved to ?location2 from ?location)}
```

Each ops-search operator has a series of conditions that determine when it is used and how it manipulates the world state to achieve a goal when applied.

Keyword	Description
pre	Conditions that must be true in the world state for the operator to be applied
del	Tuples that are deleted from the current world state when the operator is applied
add	Tuples that are added to the current world state when the operator is applied
cmd	Commands to send to external programs when performing this operator
txt	Text to output when performing this operator

Table 2 – Operator search keyword definitions.

We were able to implement the ops-search operators for our scenario very quickly, as the keywords were well defined.

When running initial, small scale testing, we saw that the plan was created with no visible delay, however as we started to increase the size of the problem scenario, the time taken to get results became an issue.

To mitigate this, we optimised and removed unnecessary tuples from the start and world states, decreasing the branching factor (Leyton-Brown, 2009) of the breadth-first search mechanism used.

6.2 Planner

The planner is a goal-driven stack-based inference engine which works by backwards chaining goals onto the stack. An example planner operator is shown below.

```
{:name move
:achieves (at ?agent ?target-location)
:when ((at ?agent ?location) (isa ?some-location location) (connects
?target-location ?door) (connects ?some-location ?door) (:guard
(correct-path (? location) (? some-location) (? target-location))))
:post ((at ?agent ?some-location) (state ?door open))
:pre ()
:del ((at ?agent ?some-location))
:add ((at ?agent ?target-location))
:cmd ((move ?agent ?target-location))
:txt ((?agent moved from ?some-location to ?target-location via
?door))}
```

Planner operators have similar fundamentals to ops-search operators with the addition of conditions that determine when an operator is used and what new goals are put on the goal stack before it can be achieved, therefore chaining the goals.

Keyword	Description
achieves	The goal state that the operator achieves (e.g. door is open after open-door operator)
when	Conditions that must be true for the operator to be used (e.g. when day is Tuesday) or to determine when one operator should be used over another that has the same achieves statement. Also used for binding.
post	Goals that need to be achieved before the operator can be applied (e.g. door is open, the archives of open-door operator)
pre	Conditions that must be true in the world state for the operator to be applied
del	Tuples that are deleted from the current world state when the operator is applied
add	Tuples that are added to the current world state when the operator is applied
cmd	Commands to send to external programs when performing this operator
txt	Text to output when performing this operator

Table 3 – Planner keyword definitions.

When implementing the planner operators, much more care had to be taken over the design of the operators, especially for the ‘when’ and ‘post’ keywords. There were also a number of issues encountered during development of the planner operators which are detailed below:

1 Single ‘achieves’ tuple limitation

As operators can only have a single tuple in the ‘achieves’ section, operators that change multiple states (such as the drop operator, which both adds an item to a location, and removes it from an agent) must be duplicated and changed slightly to allow for different ‘achieves’ tuples.

2 Infinite loop when pathfinding

When developing the move operator, the planner got into an infinite loop when attempting to move to a location that had multiple connected locations.

When the goal of moving to a location is added to the goal stack, the planner starts a depth-first search using STRIPS (Fikes & Nilsson, 1971), attempting to find a path from the current location of the agent.

To move to a target location, an agent must first move to a location connected to the target; moving to this connected location is added to the goal stack. One way of moving to this connected location is from the target location, and so an infinite loop occurs.

We resolved this problem by using a guard condition, which allows normal Clojure code execution within a tuple in the ‘when’ section of a planner operator. The guard condition used a legal move generator and breadth-first search to generate a valid path to the goal location, and ensure that the location being moved to was the next step in the path found. This approach worked, but was not as elegant as it could be in a class that is otherwise purely tuples.

3 Plan inefficiency – Ordering of ‘post’ tuples

When creating the move operator, an extra operator and careful ordering of the ‘post’ tuples were required to ensure the resulting plan was as efficient as possible. If the tuples in the ‘post’ section were inefficiently ordered, and if the agent needed to move to a location behind a locked door, the agent may walk to the door, then go back and find a key. This was not an issue with the ops-search operators.

6.3 Usability

With the operators defined for the two inference engines, a usability comparison was produced, using the factors listed below:

1 Ability to closely model the problem

As we were able to use the same tuples for each inference engine, they both worked equally well in their ability to closely model the problem domain, although the planner required multiple operators for some actions.

Due to the severe performance impact of additional tuples and operators in ops-search, non-essential tuples or operators that may more closely model the problem, may need to be removed to ensure performance is satisfactory. For example, there are no close-door or lock-door operators in our implementations.

2 Ability to change tuples

Ops-search allowed us to easily change the tuples used to model the problem if necessary, as the operators were easily defined.

When writing for the planner, as the tuples in the ‘post’ section must match a tuple in the ‘achieves’ section of another operator, operators become more tightly coupled, making change difficult.

3 Multiple goal states

Ops-search allows multiple goal states to be defined, allowing for extensibility of the scenario, for example, if the thief was to steal multiple items. This is lacking from the planner.

4 Guard conditions

The ability to use the guard condition to execute standard Clojure code inside a planner operator allows for lots of interesting applications, however it distracts from pure tuple manipulation.

7. Representation – NetLogo

NetLogo was used to visually represent the scenario and results of the inference engines due to its fast prototyping capabilities. The NetLogo model is a symbolic representation of the scenario, locations, agents, and items, displaying each state update from each operator, as shown in Figure 3.

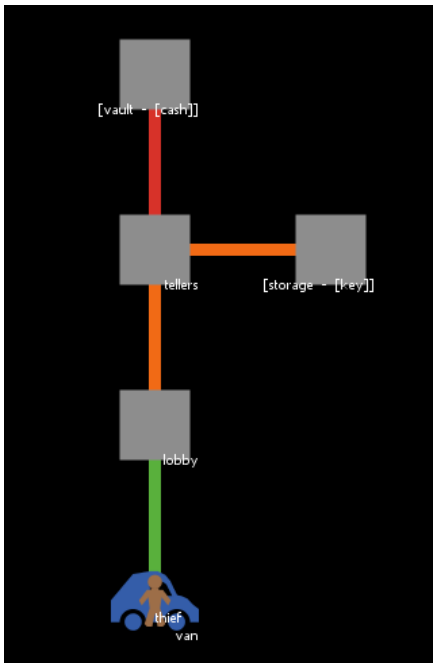

Figure 3 – Scenario as represented in NetLogo

A Java socket was used to achieve one-way communication from Clojure to NetLogo. As NetLogo lacks built-in socket communication, an extension was used, this caused a rendering issue when reading a command from the socket. This issue was resolved by adding a wait in the 'exec' command, allowing the interface time to refresh.

```
to exec.repl
  run sock2:read
  wait 0.1
  tick
end
```

The tuples in the 'cmd' section from the results of the inference engines were mapped to function names in NetLogo and written to the socket, as shown below.

```
(let [sp " "
      qt "\""
      str-qt (fn [x] (str " \"" x "\" ")) ; wrap x in quotes
      ]

 (defmatch nlogo-translate-cmd []
   ((open-door ?door) :=> (str 'exec.open-door sp (str-qt(? door))))
  )
 )
```

The NetLogo project listened to the socket and performed the actions that it received.

The static elements of the world state such as locations and doors were defined directly in NetLogo, any elements that could be manipulated, such as door state and item locations, were sent through via the Java socket. This allowed for different start states, such as new item locations, to be sent to NetLogo, as shown in Figure 4.

Figure 4 – NetLogo receiving commands from socket

Performing all complex computation, such as path finding, in Clojure allowed for separation of concerns, and reduced the time taken for the NetLogo implementation.

8. Results

Tests were carried out on the time taken for the inference engines to complete a plan under different conditions. This allowed us to analyse how performance was affected by different factors, and could help when deciding which inference engine is best suited to a particular problem.

The execution of each inference engine was wrapped in ‘time’ and ‘for’ Clojure functions. All ‘println’ and ‘ui-out’ calls were removed to reduce performance impact. One hundred iterations were performed, and the execution time for each iteration captured. The results of each test were averaged and outliers were ignored.

8.1 Performance Comparison – Move Operator Only

The first test used only the move operator within the ops-search and planner. All other operators were removed to increase performance, and any irrelevant tuples were removed from the world and start states. The initial state of the first test consisted of two locations, a starting location and a destination location, with an open door linking them together. For each subsequent test, a new open door and location was added between the start and destination location, thus increasing the number of move operations necessary to achieve the goal state.

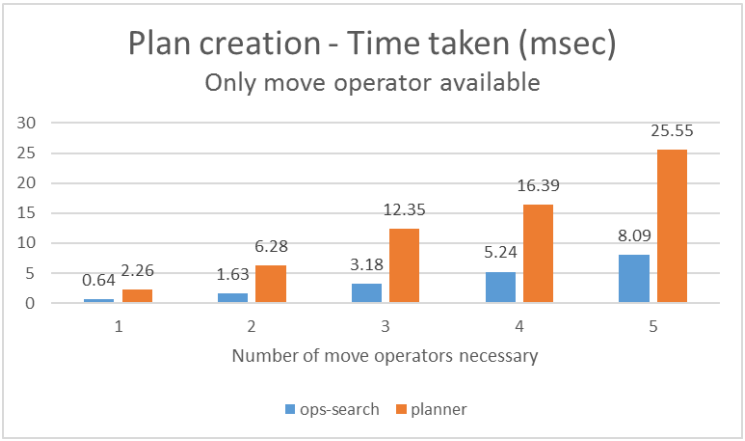

Figure 5 – Time taken to create a plan with only the move operator available.

Initial results in Figure 5 show that when one operator is available, ops-search outperformed the planner, and as the number of operations needed to reach the goal increases, the difference in time taken between the two mechanisms grows.

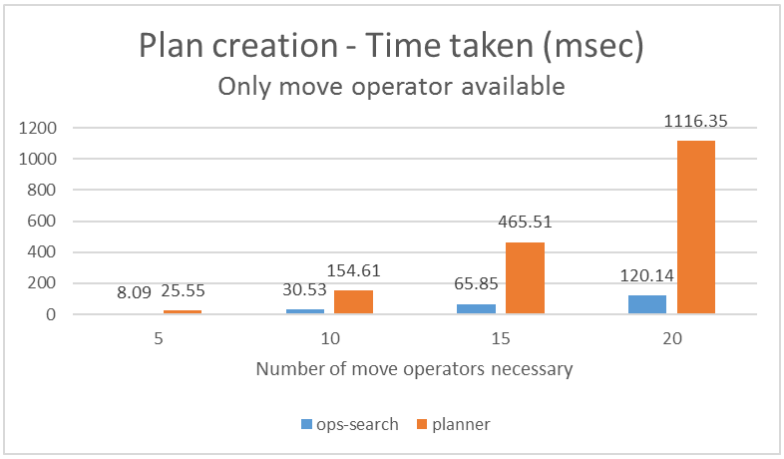

Figure 6 – Time taken to create a plan with only the move operator available.

As shown in Figure 6, with twenty move operations required to reach the goal state, ops-search was almost ten times as fast as the planner.

8.2 Performance Comparison – All Operators Available – Only Move Operator Required

The same tests were then performed with all other operators (pick-up, drop, open-door, etc.) available.

Although all operators were available, only the move operator was required to achieve the goal state, as before.

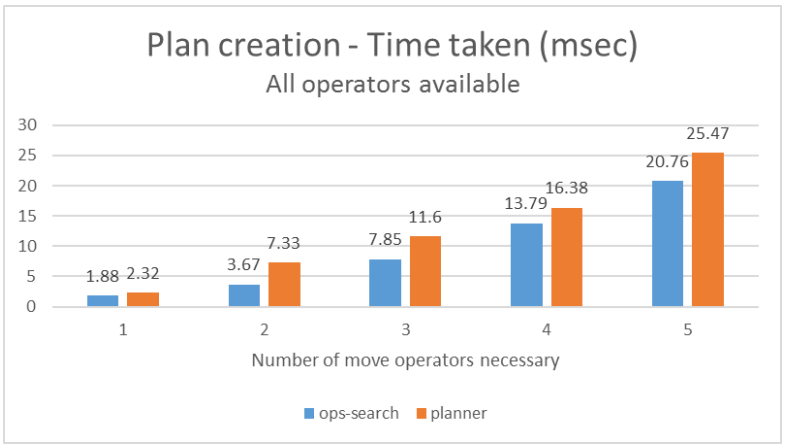

Figure 7 – Time taken to create a plan with all operators available.

Figure 7 shows that with all operators available, ops-search still outperformed the planner; however, the difference had reduced substantially.

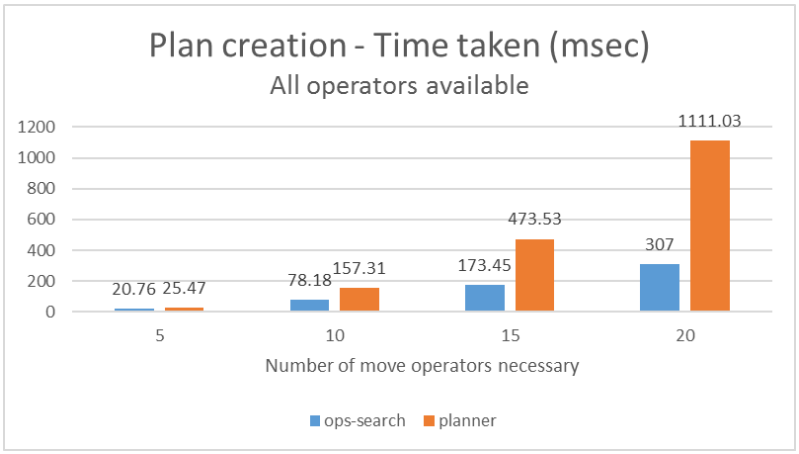

Figure 8 – Time taken to create a plan with all operators available.

As Figure 8 shows, when twenty operations were required, ops-search was four times quicker than the planner; a considerable difference, but less so than with only the move operator available.

8.3 Performance Comparison – All Operators Available – Two Operators Required

Similar tests were carried out, but with all the doors closed. This would require a second operator, open-door, to move an agent between rooms (one to open the door and one to move to the next location).

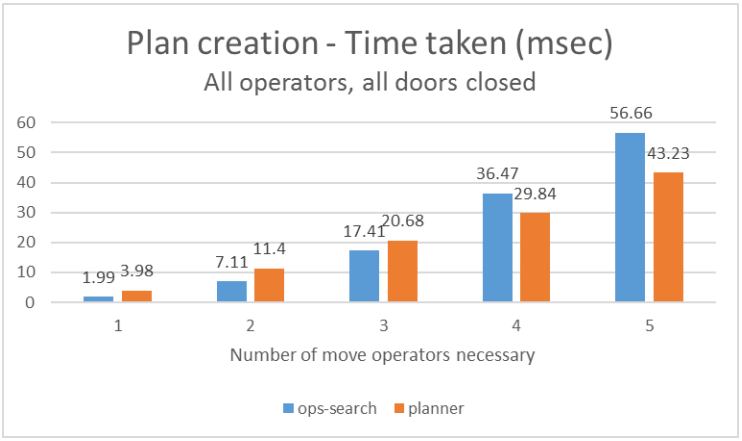

Figure 9 – Time taken to create a plan with all operators available and all doors closed.

As shown in Figure 9, the planner started to outperform ops-search after four moves were required to achieve the goal state. This was the first time the planner outperformed ops-search in our testing.

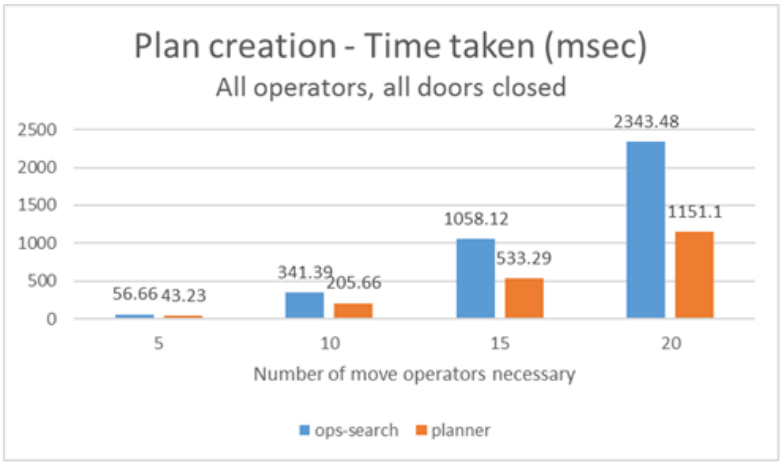

Figure 10 – Time taken to create a plan with all operators available and all doors closed.

Figure 10 shows that as the number of move operations necessary increased, the difference between ops-search and the planner continued to grow.

8.4 Performance Comparison – Operator Search

When the results of the previous tests were collated into a single graph, the effect of the additional operators and tuples on the performance of ops-search was more clearly visible.

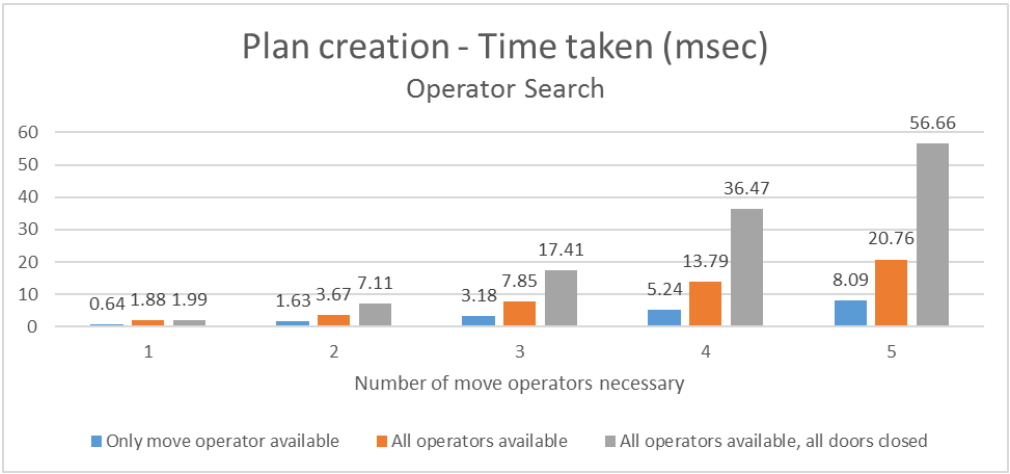

Figure 11 – Time taken to create a plan – operator search comparison.

Figure 11 shows that when all operators are available, the time taken to create a plan increases even more rapidly, but not as much as when all operators are available and all doors are closed.

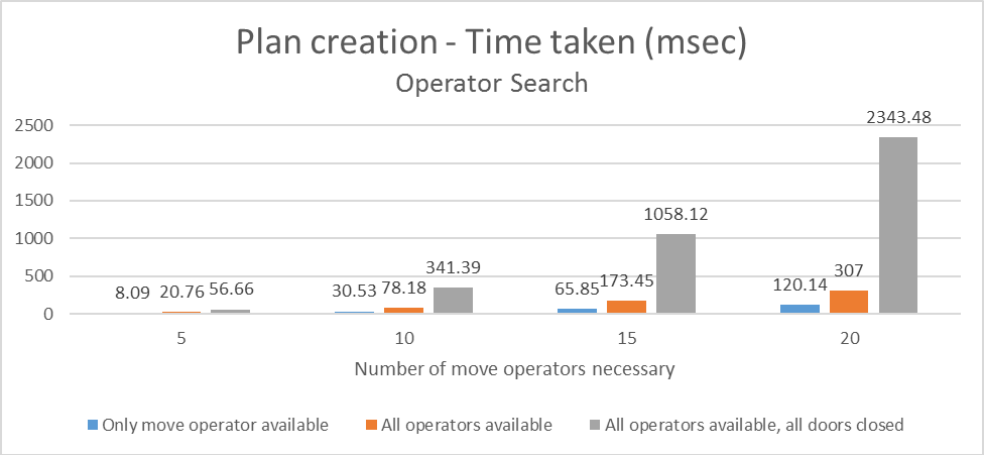

Figure 12 – Time taken to create a plan – operator search comparison.

Figure 12 shows that as the number of move operations necessary increases, the performance impact of the open-door operations become even clearer. At twenty move operations necessary, the plan takes over two seconds to complete.

8.5 Performance Comparison – Planner

Combining the results of the tests on the planner into a single graph, we found that adding the operators had almost no impact on the performance of the plan creation.

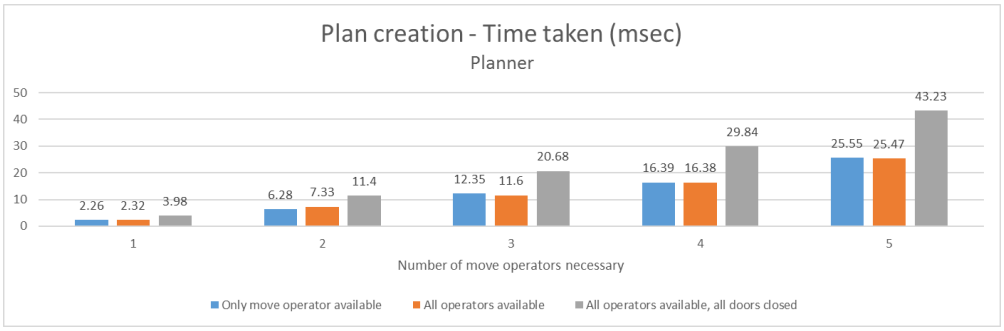

Figure 13 – Time taken to create a plan – planner comparison.

Figure 13 shows that for a small number of necessary operations, the number of available operators had very little impact on the time taken to create a plan. The extra operation, open-door, before each move operation, however, did have a noticeable performance impact.

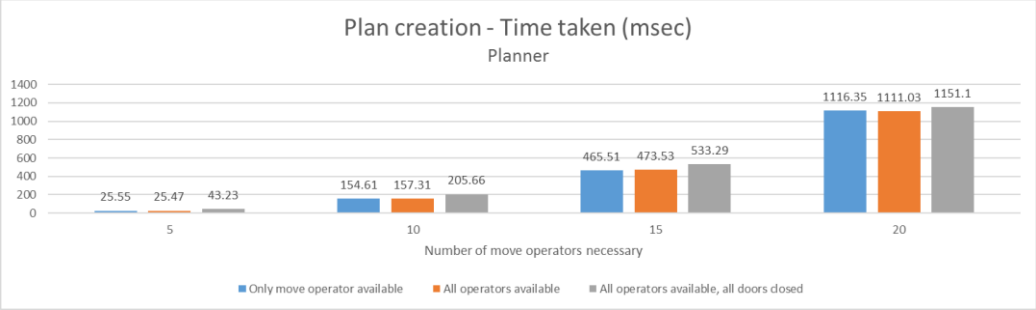

Figure 14 – Time taken to create a plan – planner comparison.

Figure 14 shows the performance impact of the open-door operations was a lot less noticeable as the complexity of the plan increased. At twenty move operations necessary, all values were very similar, regardless of the number of operators available for the move operations required to achieve the goal.

8.4 Performance Comparison – Real Scenario

Real-world problem scenarios may require multiple operators, complex world states, and lots of operations to reach a goal. Tests were performed using realistic goal states for the heist scenario, which required multiple operators. These tests started with a simple goal *(at thief lobby)*, and built up to the eventual goal state of *(in cash van)*. The tests had the full world and start state as found in Figure 3, with the key in the storage room and the cash in the vault, behind a locked door. The state of the lobby-tellers door and tellers-storage door were closed, and the entrance door was open.

The first test was a simple move of the agent from the van to the lobby, with the following command generated:

```
((move thief lobby))
```

The second test was to move the thief to the tellers, through the closed lobby-tellers-door.

```
((move thief lobby) (open-door lobby-tellers-door) (move thief
tellers))
```

The third test was to move the thief to the vault, which was behind a locked door. The key needed to unlock the door was in the storage room.

```
((move thief lobby) (open-door lobby-tellers-door) (move thief tellers)
(open-door tellers-storage-door) (move thief storage) (pick-up thief
key) (move thief tellers) (unlock-door vault-door) (open-door vault-
door) (move thief vault))
```

The final test was the full scenario, with the thief picking the cash up from the vault, and dropping it in the van.

```
((move thief lobby) (open-door lobby-tellers-door) (move thief tellers)
(open-door tellers-storage-door) (move thief storage) (pick-up thief
key) (move thief tellers) (unlock-door vault-door) (open-door vault-
door) (move thief vault) (drop thief key) (pick-up thief cash) (move
thief tellers) (move thief lobby) (move thief van) (drop thief cash))
```

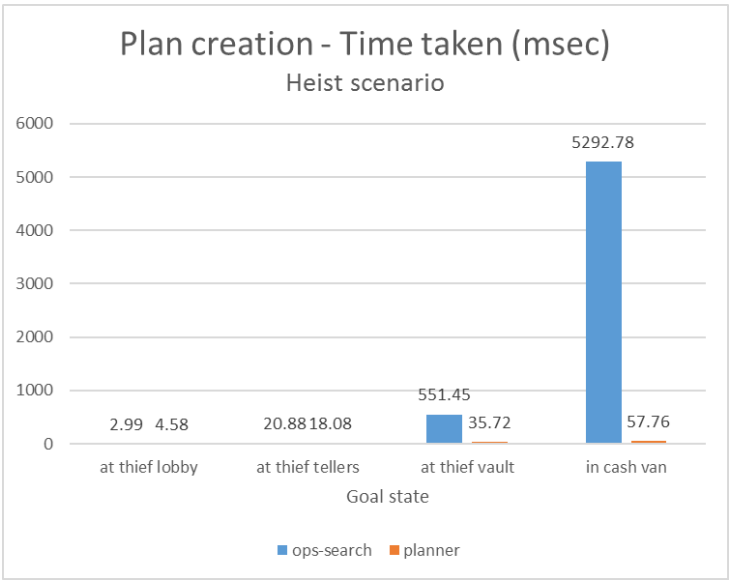

Figure 15 – Time taken to create a plan – Heist scenario.

As shown in Figure 15, for the simple goal *(at thief lobby)*, ops-search outperformed the planner. For the slightly more complex goal *(at thief tellers)*, ops-search began to take longer than the planner. For more complex goals, ops-search took far longer. It took over five seconds for ops-search to create a plan for the full scenario *(in cash van)*, almost one hundred times slower than the planner, which created a plan in under sixty milliseconds.

9. Critique

Although we feel our research methods are valid and useful, there were a number of issues that could have been improved.

9.1 Performance Testing

Although extensive performance testing was conducted, and outliers ignored, the tests were performed on a desktop machine with other processes and services running in the background, which could have impacted the results.

No baseline was determined for the tests, and no performance profiling tool was used. A profiling tool would have given extra information, such as memory usage and CPU cycles required. This would have also isolated resource consumption from other operations on the machine.

When performance testing with all scenario operators available, the planner has two more operators. This could impact the performance of the planner, but also reflects the reality that the planner occasionally requires multiple operators to perform the same job as ops-search.

9.2 Planner – Breadth-First Search

In the move operator for planner, we used a guard condition to ensure we moved to a location on the correct path. This guard condition used breadth search, but this could perhaps be improved by using a more efficient search algorithm, such as A* (Red Blob Games, 2014).

This could be the reason we saw the planner perform worse than ops-search for the tests that required lots of move operations. For more accurate results, a test could have been performed using operators that do not have the guard condition.

9.3 Planner – Features Not Investigated

The planner has the ability to protect goals using the *protected* keyword, and negate a condition using the *not* keyword. These features could be very useful, and offer another advantage over ops-search, but were not necessary for our heist scenario.

9.4 Storage Used

The storage space used by each inference engine differs, and could have been analysed. This would have made an interesting comparison to the performance of each engine, and could help when deciding which tool to use for a given scenario.

9.5 Clojure Storage Limitation

As the JVM has a fixed allocation of stack and heap space, a stack overflow may occur when creating a plan for complicated goals with ops-search, as the breadth-first search runs out of stack space.

The planner has a configurable limit on the number of STRIPS loops permitted; this can be used as a safeguard to ensure stack overflows do not occur.

10. Further Investigations

We investigated the ops-search and planner inference engines using a heist scenario and tested their performance. Although we were able to draw conclusions from our results, further investigation could provider more evidence to reinforce our claims.

The heist scenario was designed to be extensible; possible extensions on the model could include a security guard, alarm, or camera that the thief must avoid, the addition of more keys that unlock different doors, or a drill that could be used to open the vault door.

Other inference engines, such as those consisting of Belief Desire Intentions (BDI) agents, could also be evaluated. These may be better suited to certain scenarios.

As mentioned in the critique, evaluation of the storage space usage of each inference engine would provide useful data and an interesting comparison to their performance.

11. Conclusion

Reflecting on the performance results and ease of implementation of the two inference engines, we drew the following conclusions.

11.1 Ops-search

We found implementing the ops-search operators very easy, as the sections of each operator were well defined, and did not depend on any other operators. The performance, however, became an issue on anything other than the simplest of scenarios. In the full heist scenario, for example, ops-search took over five seconds to create a plan.

11.2 Planner

The implementation of the planner was much more difficult than ops-search, due to the tight coupling of the operators, and the fact we had to have multiple operators for some operations (such as move and drop). We also ran into trouble with infinite loops when creating the move and pathfinding operators. Fortunately, the ability to execute Clojure code via a guard condition was able to resolve the issue.

Once these issues were resolved, however, the planner was able to create plans much faster than ops-search in the heist scenario (see Section 8.4).

Although the planner performs worse than the ops-search in our performance tests of multiple move operators, this may be due to the breadth-first search used in the guard condition. The conclusion here would benefit from performing tests on operators that do not require the guard condition.

11.3 Summary

In summary, we recommend that for simple or small scale planning, or situations where time is not an issue, ops-search is used over the planner. Although ops-search is relatively slow due to the use of breadth-first search; the ease of implementation, broad usage possibilities, and simplicity are commendable.

In situations that are time-sensitive, the planner is recommended due to its significant speed advantage.

12. Glossary

Backward Chaining

> When using Backwards Chaining, the desired goal is posted to the goal stack. The system creates a plan by checking to see if the goal is currently satisfied in the existing knowledge base (world state). If the goal is not currently satisfied, the system searches for an operator that can satisfy the goal and posts that operator's preconditions to the goal stack. This process is repeated until the conditions for an operator are satisfied in the world state, at which point the goal is popped off the goal stack and the operator is applied, adding new facts to the world state. (Sharma, Tiwari, & Kelkar, 2012)

Breadth-First Search (BFS)

> Used in tree and graph traversal, the BFS algorithm starts at the root of a tree and explores every node at a given level before moving onto the next level. (Robin, 2009)

Forward Chaining

> Forward Chaining is a method used by Inference Engines. Applying rules against a knowledge base (world state), based on a condition (antecedent). If the condition evaluates to true, the rule is applied and new facts (consequents) can be added to the knowledge base. This process continues until there are no more rules that can be applied. Forward chaining is an exhaustive search and may explore and apply rules that do not lead towards a defined goal. (Sharma, Tiwari, & Kelkar, 2012)

Inference Engines

> Given a knowledge base, Inference Engines apply search techniques and control strategies to make decisions. Inference Mechanisms are commonly used within Expert Systems to aid with problem solving.

Legal Move Generator (LMG)

> A function that takes a given state and generates all possible successor states. Often used in conjunction with breadth-first search to limit the number of nodes expanded.

Means End Analysis (MEA)

> MEA is a technique used for goal-based problem solving. It works by applying operators (rules) to an initial world state. Each operator applied may alter the current world state. Operators are chosen based on which operation will reduce the difference between the current state of the world and the goal state required.

STRIPS

> STRIPS (STanford Research Institute Problem Solver) is a problem solver that looks for a sequence of operators that can be used to manipulate a world state, it uses Means End Analysis and attempts to plan a suitable solution using a goal-satisfying model. (Fikes & Nilsson, 1971)

13. References

Fikes, R. E., & Nilsson, N. J. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence 2*, 189-208.

JetBrains. (2015, November 2). *Clojure plugin for IntelliJ IDEA*. Retrieved from GitHub: https://github.com/JetBrains/la-clojure

JetBrains. (2016). *IntelliJ IDEA*. Retrieved from JetBrains: https://www.jetbrains.com/idea/

Leyton-Brown, K. (2009). *Breadth-first Search; Search with Costs.* Retrieved from CPSC 322 - Introduction to Artificial Intelligence (Term 1, Session 101, 2008-09): https://www.cs.ubc.ca/~kevinlb/teaching/cs322%20-%202008-9/Lectures/Search3.pdf

Lynch, S. (n.d.). *ops-search*. Retrieved February 18, 2016, from http://s573859921.websitehome.co.uk/pub/clj/tools/ops-search.htm

Lynch, S. (n.d.). *planner*. Retrieved February 18, 2016, from http://s573859921.websitehome.co.uk/pub/clj/tools/planner.htm

Newell, A., Shaw, J. C., & Simon, H. A. (1959). *Report on a General Problem-Solving Program.* Santa Monica: The RAND Corporation.

Rajeev, S., & Krishnamoorthy, C. S. (1996). *Artificial Intelligence and Expert Systems for Engineers.* CRC Press. Retrieved from https://books.google.co.uk/books?id=zHbxtrzffKcC&pg

Red Blob Games. (2014). *Introduction to A**. Retrieved from Red Blob Games: http://www.redblobgames.com/pathfinding/a-star/introduction.html

Robin. (2009, December 16). *BREADTH FIRST SEARCH*. Retrieved from World Of Computing: http://intelligence.worldofcomputing.net/ai-search/breadth-first-search.html

Sharma, T., Tiwari, N., & Kelkar, D. (2012). STUDY OF DIFFERENCE BETWEEN FORWARD AND BACKWARD REASONING. *International Journal of Emerging Technology and Advanced Engineering*, *2*(10), 271-273. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.414.5336&rep=rep1&type=pdf