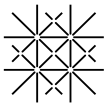

University of Basel

Best-first search with trial-based Open-list

Bachelor

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence
https://ai.dmi.unibas.ch/

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Thomas Keller

Rasmus Jensen
r.jensen@stud.unibas.ch
17-947-664

15.07.2022

Abstract

Monte Carlo search methods are widely known, mostly for their success in game domains, although they are also applied to many non-game domains. In previous work done by Schulte and Keller[4], it was established that best-first searches could adapt to the action selection functionality which make Monte Carlo methods so formidable. In practice however, the trial-based best first search, without exploration, was shown to be slightly slower than its explicit open list counterpart. In this thesis we examine the non-trial and trial-based searches and how they can address the exploitation exploration dilemma. Lastly, we will see how trial-based BFS can rectify a slower search by allowing occasional random action selection, by comparing it to regular open list searches in a line of experiments.

Table of Contents

1 Introduction

Automated planning is a branch of AI, which envelops the searching of strategies and actions for autonomous entities to execute, these being machines or humans.
The context in which these strategies are realized vary, from deciding on a game strategy in the game of chess, to the automated conduction of unmanned vehicles.

Automated planning includes many different sub-areas of planning. Searching for strategies in chess, for example, is part of classical planning, whereas driving an unmanned vehicle would constitute a conditional planning problem. Classical planning searches are further split between forward and backward searches.

In this paper, we have a look at a certain group of forward searches, namely best-first search, which make use of a heuristic enhancement. We will compare best first search (BFS) with trial-based open list (TOL)[4] to BFS with an explicit open list. We will extend the Fast-Downward classical planning system, which was founded and is being maintained by Malte Helmert[3], with TOL searches, and we will be comparing them to the existing epsilon-greedy open list implementation of Valenzano et al.[5]. We will talk about the fundamental issue of balancing exploration and exploitation, and we will view several methods which seek to address it. Lastly, we will discuss the results, analyzing the best searches and coming up with improvements.

2
Background

2.1 Automated Planning

Automated Planning is a broad term used to describe the deduction of a strategy to be executed by an autonomous agent. An autonomous agent refers to any entity which is able to execute (directed) actions without any assistance. With the help of automated planning, we are able to give machines directives, which would otherwise be infeasible due to the task complexity.

The act of deducting a strategy is called a search. The search is a simulation of an agent performing actions in an environment. The search will play out scenarios in order to determine the best strategy for the agent to perform. The environment, in which the search is conducted is called a state space. The state space is a mathematical representation of an environment, as well as container for actions the agent can execute. The state space additionally contains the task to be completed by the search, in the form of an initial state and a set of goal states.

Definition - State space A state space is a tuple $\mathcal{S} = (S, A, c, T, s_0, S*)$ where

- S is the set of all states.
- A is the set of all actions.
- c is the cost function $c : A \to \mathbb{R}_0^+$
- $T \subseteq S \times A \times S$ is the relation that describe all possible transitions within the state space.
- $s_0 \in S$ is the initial state.
- $S^* \subseteq S$ are the goal states.

An environment which is completely known can be described with a state space, as we can formulate and numerate all of its aspects. If the environment is completely known, and for $(s, a, s') \in T \to (s, a, s'') \notin T$, meaning that an action may not have several transitions

from the same state, then the resulting state space is deterministic. Deterministic state space searches are what the subbranch of automated planning, classical planning, concerns itself with. The game of checkers is an example of a domain/environment which has a deterministic state space representation.
Unknown and partially unknown environments usually have non-deterministic aspects. An example of this could be the a robotic agent on earth, trying to walk across a busy highway. It might have a sensor to recognize approaching vehicles, but it cannot predict when exactly a vehicle will arrive, to precompute a plan. Instead, the search for a strategy needs to be done live(online), using the sensors to track oncoming traffic. Since we will be working within known environments, we can fully describe them in a state space, and we can compute these strategies offline.

State space representation State spaces with many states cannot realistically be represented by humans using the definition above, as each state needs to be declared individually. Instead, we can represent the state space compactly by defining a set of state variables V, where each state is represented as a tuple of variable assignments.

This means a state, i.e. a **total assignment**, is defined by the values of the variables $v \in V$. A total assignment α for a set of variables V is a tuple $\alpha = (d_1, ..., d_{|V|})$, for which every variable $v_i \in V$ is assigned a value $d_i \in dom(v_i)$ in the domain of v_i. A set of states can be defined by a **partial assignment**, which is an assignment, for which only a subset of the variables in V are given values. Partial assignments simplify the definition of goal states and setting non-trivial action constraints.

An action a is **applicable** in state s, if there exists a state $s' \neq s$ for which the transition $(s, a, s') \in T$. In that case s is called **parent** or **predecessor** of s', and s' is called **child** or **successor** of s.
A state s is **reachable** from state s_0 if there exists a sequence of transitions in T $< t_0, t_1, ..., t_n >$, so that $t_0 = (s_0, a_0, s_1)$ and $t_n = (s_n, a_n, s)$.
A **path** from s_0 to s_{n+1} is a sequence of actions $< a_0, a_1, ..., a_n >$, so that $t_i = (s_i, a_i, s_{i+1})$ and a_i is applicable is s_i, for $i \in \{0, ..., n\}$.

Action Languages Action languages are programming languages, which are used in connection with planning tasks. They utilize the compact form of state space representation, using variable assignments or predicates. A predicate can be viewed as a binary assignment, since it is either true, or false.

PDDL is one of the action languages used in the Fast-Downward planning system. Throughout the years, the language has evolved, creating a family of languages with differing levels of detail and expressivity.

Example - Blocks world domain

(define (domain BLOCKS)

```
(:requirements :strips)
(:predicates (on ?x ?y)
  (ontable ?x)
   (clear ?x)
   (handempty)
   (holding ?x))
(:action pick-up
   :parameters (?x)
   :precondition (and (clear ?x) (ontable ?x) (handempty))
   :effect(and (not (ontable ?x))(not (clear ?x))(not(handempty))(holding ?x)))
(:action put-down
   :parameters (?x)
   :precondition (holding ?x)
   :effect (and (not (holding ?x))(clear ?x)(handempty)(ontable ?x)))
(:action stack
   :parameters (?x ?y)
   :precondition (and (holding ?x) (clear ?y))
   :effect (and (not (holding ?x))(not (clear ?y))(clear ?x)(handempty)(on ?x ?y)))
(:action unstack
   :parameters (?x ?y)
   :precondition (and (on ?x ?y) (clear ?x) (handempty))
   :effect (and (holding ?x)(clear ?y)(not (clear ?x))(not (handempty))(not (on ?x ?y)))))
```

Problem domain The domain defines the universal aspects of the search problem. In this part are defined the types of objects existing in the universe, as well as predicates for these types.

This is also where all actions are defined. An action has a set of predicates, called the precondition, and another set of predicates containing the effects of this action. An action can only be taken, when the precondition is satisfied. Lastly, the cost of the action is set, if needed.

An example of this is the Blocks world, where there are predicates such as **on x y**, **ontable x**, **clear x** or **holding x**. x and y are placeholders for certain object types. In the case of blocks world, there exist only blocks. The actions are pick-up, put-down, stack, unstack.

Problem definition The problem definition now contains a set of domain objects and their types, and the initial state of the problem as a set of predicates. We also define all goal states that lead to a solution of this search problem, using predicates.

State space functionalities We will be viewing a broad spectrum of different problem domains, so in order to give a sensible overview of search algorithms, we define these universal black-box functionalities for state spaces.

- **init()** - generate the initial state s_0
- **succ(s : State)** - find all successor states
- **par(s : State)** - retrieve parent state
- **cost(a : Action)** - calculate the cost of a
- **is_goal(s : State)** - check if s is a goal state
- **trace_path(s : State)** - trace a path from s back to s_0

2.2 Forward Search

A forward (chaining) search, is an algorithm which conducts a search for a goal state, starting at the initial state s_0. The planner can generate the initial state, and begin the search there. The planner **expands** the current state, meaning it finds all applicable actions from the current state, and it then **generates** the successor states. The successors are **evaluated** and then added to the state space graph. A successor state is chosen for expansion, and the steps are repeated, until a goal state is found. If a goal state was found, the planner traces a path back to the initial state, in which case it reports a successful search, and returns the sequence of legal actions that lead to a solution.

Algorithm 1 Example of a tree search algorithm

```
open = [ ]
open.add(init())
while open is not empty do
    s ← open.removeElement()
    open.add(succ(s))
    if is_goal(s) then
        return trace_path(s)
    end if
end while
```

2.2.1 Tree Search

A tree search is a search algorithm, which keeps an open list. In every step, it will remove an state from the open list, and add all the state's successors to the open list. If the state is a goal state, it traces back and returns the path from s_0 to s. The way in which we maintain the open list influences the order of states visited and therefore the path to the goal. There are a great variety of open list implementations, most notably the priority queue, for which specifically the min-heap is a very efficient way of removing (or popping) a prioritized element. However there are other sorts of open lists aswell.

2.2.2 Graph Search

A graph search is a specific type of tree search, which additionally maintains a closed list. This closed list serves as a way to ignore previously visited states, by preventing states that

are in the closed list to be added to the open list. Beyond the difference in utility, searches are usually faster when keeping a close list, at the cost of additional memory overhead for maintaining it. When using closed lists, the option for reopening states is presented. A state can be removed from the close list, if the state was found on a cheaper path. Usually a graph search with reopening will return slightly better solution costs that one without.

2.3 Data structures

2.3.1 Search Node

A search node data structure and is a container for the state, it is used instead of a state in the state space search. It can be described as tuple $n = \langle s, p, \mathcal{N}, f \rangle$, where s is the state, p is the parent, $\mathcal{N}$ are the children nodes, and f is the value estimate. Using this container, the planner can traverse the generated graph.

In most cases, having a lookup table for states and instead wrapping a state ID, will save memory when initializing new nodes.

2.3.2 Open list

The open list is a data structure, which generally contains nodes that are yet to be expanded. From the open list are picked the future nodes to be expanded. Depending on the search, a different type of open list is used. For example, depth first search uses a stack, to order the elements, whereas breadth first search uses a FIFO (first-in first-out) queue. These searches base their queue order on the order of generation, and not on any evaluations, so they are named uninformed searches.

Heuristic search Heuristic search is a term for a group of search algorithms that order the open list based on a heuristic function. The heuristic function maps assigments to a value in $\mathbb{R}_0^+$. The heuristic value represents the quality of a node, in terms of the distance to the goal. It is important to note that the heuristic value is not an exact measurement of the distance, as the path to the goal is not known at runtime. However many heuristics are reliable to an extent, some more than others.

The heuristic and uninformed searches can be implemented both as tree as well as graph searches.

2.3.3 Closed list

The closed list is a list containing nodes which have already been expanded in the search.

The closed list often improves the speed of the algorithm, as it prevents the search engine from considering previously expanded nodes. There are searches where duplicate elimination is not needed, or even wanted. Keeping a closed list requires a lot of memory. Close lists are usually implemented as hash sets.

2.4 Best first search

Best-first searches have been around since the 1960s, with A* being created as part of the Shakey projects task planning system. However, best-first searches to this day are some of the most efficient and consistent heuristic searches discovered.

Best first searches are a type of search algorithm, that explore the graphs by expanding the most promising nodes first. The value of a node is predetermined by an evaluation function f, which is based on the current path cost from the initial state g, and the heuristic function h, where h approximates distance from a goal state.

Different BFS algorithms have different evaluation functions. For example A* uses the evaluation function $f = g + h$ to order the open list. In a sense A* tries to optimize solution path cost and chooses the path with the presumably cheapest path. With the correct heuristic, A* can be provably optimal, meaning A* returns an optimal solution if it exists. Greedy BFS, on the other hand, uses the evaluation f = h, which only depends on the heuristic value. When expanding a node, GBFS focuses on the shortest *remaining* path to the solution, approximated by the heuristic, and doesnt care about the path already traveled. A search is greedy, if the evaluation does not depend on past knowledge, such as g, being the current path length. Hence the connotation "greedy".
In the following we will show some greedy best first algorithms ($f = h$), with reopening, as this additional feature improves solution costs even further, at the cost of some memory complexity.

Algorithm 2 BFS

```
openlist ← new MinHeap < SearchNode >          ▷ Order the min-heap using h,g
s ← init()
if h(SearchNode(s)) < ∞ then
    openlist.add(SearchNode(s))
    closed ← new HashMap < State >             ▷ Hash-map storing closed nodes
end if
while not openlist.empty() do
    node ← openlist.pop()                                          ▷ Pop minimum
    if not closed.contains(node.state) or g(node) < node.get_g() then
        node.get_g() = g(node)          ▷ Where g returns the distance from root node
        if is_goal(node.state) then
            return trace_path(node)
        end if
        for ⟨a, n'⟩ in succ(node) do
            if h(s') < ∞ then
                n' = make_node(n, a, s')
                openlist.add(n')
            end if
        end for
    end if
end while
```

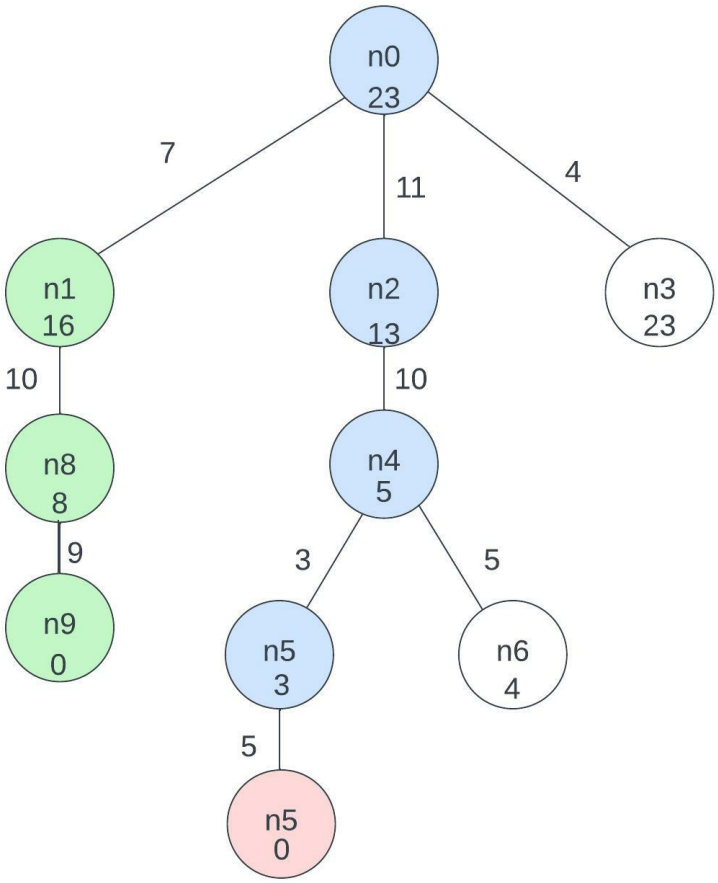

Figure 2.1: GBFS (Algorithm 2) - The tree shown is generated by the GBFS algorithm. The blue nodes show expanded nodes. The red nodes represent goal states. The green path showcases the shortest path to goal. The value on edges are action costs, and the values within nodes are the heuristic values. As mentioned earlier, GBFS expands the most promising nodes first. After GBFS has expanded n0 and generated n1, n2 and n3, it chooses to expand n2 first, as its f-value is the lowest $h = 13$. After generating and expanding both n4 and n5, GBFS reveals a goal state n5. It is clear, looking at the highlighted path in green, that the path chosen by GBFS was not the optimal one. Unlike A*, which looks for an optimal solution in terms of the path cost, GBFS looks for a solution with the least amount of actions taken, by purely focusing on the h-value.

Algorithm 2 and the Figure 2.1 depicted above shows the GBFS algorithm as it generates the tree and traverses it. The blue nodes show expanded nodes, and red nodes represent goal states. The optimal path to a goal state, in terms of action cost, is highlighted with green. We can see in this case the algorithm ordered the open list using the minimum of h, and we can see the expanded nodes are, in each their turn, the lowest f-values for $f = h$. GBFS will always try to take the *minimum* amount of steps needed, towards the goal. If there were a goal state with heuristic value 0 (like n5) as a child of the root node, the greedy BFS will choose this path, no matter the cost of the action leading to it.

Algorithm 3 EGBFS(eps)

```
openlist ← new MinHeap < SearchNode >                    ▷ Order the min-heap using h
s ← init()
if h(SearchNode(s)) < ∞ then
    openlist.add(SearchNode(s))
    closed ← new HashMap < State >                       ▷ Hash-map storing closed nodes
end if
while not openlist.empty() do
    e ← drand(0, 1)                                      ▷ random double in [0,1]
    if e <= eps then                                     ▷ Exploration
        node ← openlist.getRandom()
    else                                                 ▷ Exploitation
        node ← openlist.pop()                            ▷ Pop minimum
    end if
    if not closed.contains(node.state) or g(node) < node.get_g() then
        node.get_g() = g(node)                           ▷ Where g returns the distance from root node
        if is_goal(node.state) then
            return trace_path(node)
        end if
        for ⟨a, n′⟩ in succ(node) do
            if h(s′) < ∞ then
                n′ = make_node(n, a, s′)
                openlist.add(n′)
            end if
        end for
    end if
end while
```

A drawback of heuristic search however, is that the heuristic cannot perfectly predict the distance from goal, it often serves as an approximation. The inconsistencies can lead to suboptimal searches, which increase search time and memory costs. In many cases this can even lead to suboptimal strategies being realized. As a remedy, one can introduce a factor of exploration, as in a feature of randomness. A well-balanced search would *exploit* the heuristic whenever it is predicting correctly, and disregard it (instead *exploring* the state space) otherwise. The goal of the epsilon-greedy BFS (or EGBFS) is to balance heuristic exploitation with random action exploration. The version of the EGBFS used in the experiments for comparison, have been based on the work of Valenzano et al.[5]. It uses *knowledge-free exploration*, in other words it uses an exploration parameter ϵ to choose between random and greedy open-list selection.

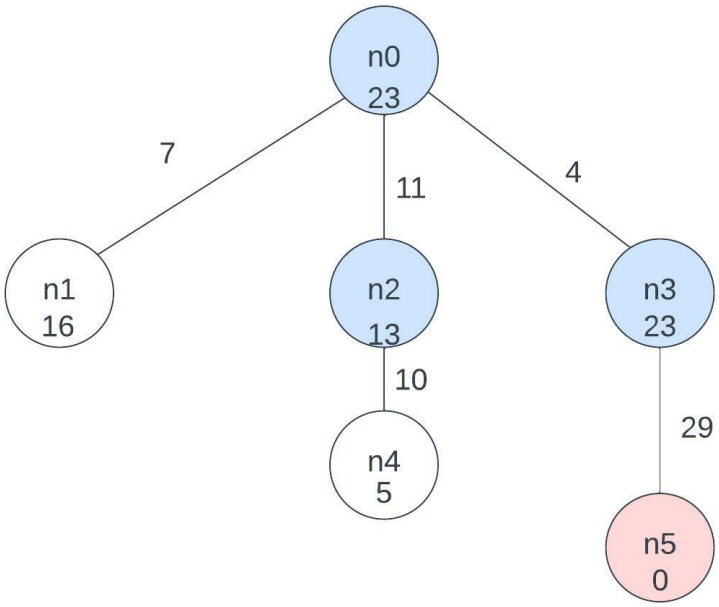

Figure 2.2: EGBFS (Algorithm 3) - For EGBFS, we start the tree off in the same fashion as for GBFS. Since EGBFS is a greedy search, the open list order is based on only $f = h$ the heuristic. First EGBFS expands the root node n0, and generates n1, n2 and n3. It then expands n2, because it has the smallest f-value. However, in the next step, the search chooses to explore a little, so a random node is picked from the open list, namely n3. It was not supposed to be picked, but due to the introduced exploration, we expand n3 and find a goal node n5 as its successor.

Algorithm 3 and the figure 2.2 depicts an epsilon greedy variant of the BFS class. The greediness is visible due to nodes being picked based only on the heuristic, with some probability of a random element being chosen from the open list. This allows for new paths to be found, that the heuristic might deem suboptimal.

3
Contribution

3.1 Explicit and Trial-based open lists

So far, all the open lists we have viewed were examples of explicit open lists. However, we can simulate the popping of an element by traversing the generated search tree, until we reach a node contained in the open list. The traversal of the tree until reaching an open node, and the subsequent expansion is called a *trial*.

3.1.1 Explicit open list (OL)

The open list contains static tuples of nodes with f-values. The next open node to expand is selected by retrieving the top element in the list, as it contains the node with the smallest value.

This is primarily done with priority queues, for which the min-heap is the best candidate. The min-heap's primary purpose, is to provide quick removal of the minimum element. It does so in $\mathcal{O}(1)$, and manages element insertion in $\mathcal{O}(\log n)$, where n is the amount of nodes in the min-heap.

3.1.2 Trial-based open list (TOL)

In the trial-based selection, the process begins at the root node and an action is continuously chosen based on the selection policy, until a previously unexpanded (i.e *open*) node, in the frontier, is reached. The frontier of a tree is the set of leaf nodes in the tree. In order for a node in the frontier to be picked for expansion, each action on the path from n_0 to the leaf node must be selected, in order.

The Algorithm 4 below shows a version of a trial-based open list algorithm, **with** exploration. The algorithm selects the best-value successor with probability $(1 - eps)$, otherwise picks a random successor. For TOL, this algorithm is used repeatedly, until a leaf node is chosen, in which case TOL expands the node.

Algorithm 4 Select_Action(eps)

```
node ← SearchNode(init())
r ← random_double(0, 1)
if r <= eps then
    node ← select_random(node.get_children)          ▷ Select random successor
else
    node ← choose_min_node(node.get_children)        ▷ Child with min. f-value
end if
return node
```

In comparison to keeping an ordered list and popping the minimum, the trial-based open list needs to compute a path to the frontier every time a node is to be expanded. For maximum search tree depth d and maximum degree c(i.e the maximum amount of children to one parent), a trial takes $\mathcal{O}(d \log c)$ time, as we need to choose a successor from the list of children in $\mathcal{O}(c)$, at most d times. The generation of a new node and its addition to the tree takes constant time, however, which adds a constant factor $\mathcal{O}(c)$ to the overall time. Overall, for an explicit open list with n number of elements, the time to pick a element and adding its (at most c) successors to the list is $\mathcal{O}(c \log n)$, whereas a trial needs $\mathcal{O}(dc)$ time. The tree depth is mostly in $\mathcal{O}(\log |S|)$, S being the set of states in the state space. However in extreme circumstances, when the tree has a very long chain of nodes, it is worst case in $\mathcal{O}(|S|)$, and therefore the overall worst case time compl. for 1 trial in TOL is $\mathcal{O}(c|S|)$, and 1 iteration for OL is $\mathcal{O}(c \log n)$. The maximum degree c is usually smaller than the tree depth d. The explicit open list is quicker in general, however with a high maximum degree d, trial-based open lists can surpass the explicit version. The slight overhead as shown in Schulte and Keller's paper[4], becomes apparent when working on larger sets of problems, as the trial-based version of GBFS has lower task coverage (number of successful tasks run) than the explicit GBFS. This does not come as a surprise, as with large state spaces, trial-based searches will mostly lag behind.

The immutability of the open list means that it is impossible to update an entry in the list. Instead, a new entry with the updated value must be entered. This makes the open list contain older versions of potentially expanded nodes, creating additional memory and time overhead. For the trial-based action selection, the entire tree needs to be constantly kept updated, as when back propagating utility values. We need to backup the frontier's utility values, as we need them for the action selection policy. This means the selections made are dynamic and versatile, as any changes in the tree will immediately affect the expansions. It does come with the price of additional time complexity, as the backup requires to move up the tree, until reaching the root node. With maximum tree depth d, this requires an additional $\mathcal{O}(d)$ time. When reopening, we additionally need to forward propagate the g-value to the sub-tree with current node n as root, because the path from initial node to n was found on a cheaper path. This adds a worst case time overhead of $\mathcal{O}(|S|)$, as the subtree can have a size of up to $|S| - 2$.

Searches which utilize the trial-based action selection are called Trial-based heuristic tree searches (THTS). For the ease of notation later on, we will call them trial-based open list

searches (TOL) instead. TOL are not necessarily tree searches, such as defined in the previous chapter. The word tree in this case refers to the maintenance of a tree structure to enable tree traversal and action selection.

Although TOL-GBFS is slightly slower than GBFS, as we will see with the use of exploration, the tree structure used in TOL is able to use exploration to a much larger extent.

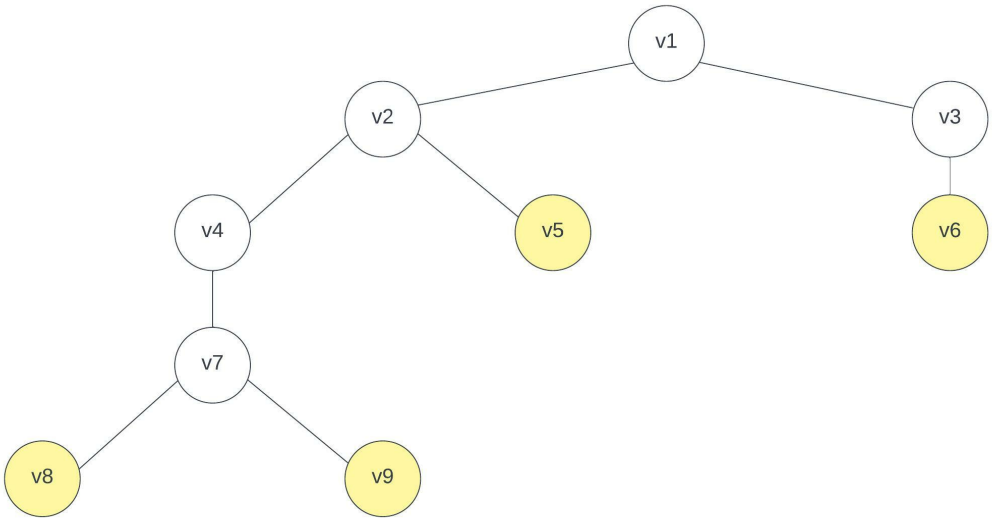

Figure 3.1: The frontier in the tree is highlighted in yellow. The frontier is the set of all leaf nodes in the tree. With , the trial begins at v1, and picks an action according to the policy. This continues until one of these highlighted, leaf nodes is reached. The search algorithm then expands the node, and generates its successors. These successors then are then added to the tree, and they become the new leaf nodes.

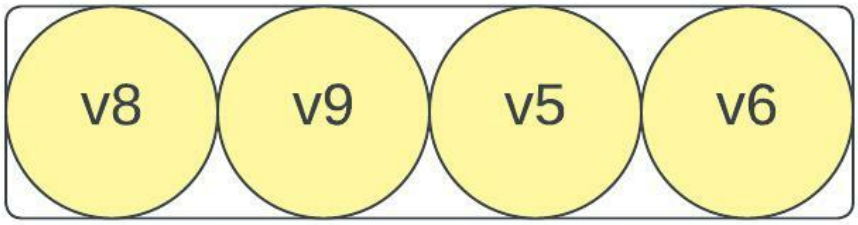

Figure 3.2: This is an example of an open list with the same search setting as for the tree above. All the nodes, which were highlighted in yellow, are contained within this list. An open list search would pick a node in the list according to the policy, and expand it just the same. But instead of adding the successors to a tree, we add them to the list itself.

The frontier in a TOL search tree, is equivalent to the nodes contained in the open list. In the open list, the nodes are ordered according to the evaluator values. In the trial-based open list, the evaluator values are used when choosing an action

3.2 On balancing exploration and exploitation

Having showcased TOL-GBFS and the GBFS having explored the inner workings of both algorithms, we proceed to compare them in terms of exploration. Exploration is knowledge-free, when the exploration parameter is fixed.

3.2.1 Knowledge-free exploration

The open list contains all leaf nodes, which still need to be expanded. Choosing a random element in the open list spreads the probability of being chosen equally among all the nodes, because the open list doesnt keep track of where its node elements are located in the tree. The removal of an element in an open list is a one-time action and can be considered atomic with respect to the search algorithm.

In contrast, TOL (trial-based open list) can randomize the choice of successor at most d times, d being the maximum tree depth. Starting a trial from the root, we can in each step choose a random successor among the set of children, until we reach a leaf node. Trial-based open list selection has multiple iterations, in which we can enforce exploration, and so the way in which we want the search to explore the tree can be conveyed in more detail. For TOL, we might be choose a random action, and then follow the most promising path in the leftover subtree for a while, only to later on choose another random successor.

An arbitrary node selection in the open list is a one-time, completely random endeavor. The TOL adaptation of EGBFS seems to explore alot more than the standard implementation, even with same exploration coefficient. This can be seen in the example figure, Figure 3.3. This is because the exploration may happen at each iteration of the action selection, unlike with the open list. It makes exploration with TOL more likely to select nodes that are in upper layers, rather than the lower (deeper) layers of the tree. The deeper the tree, the more TOL with fixed (or knowledge-free) exploration reduces the probability of the leaf node with the best f-value being chosen. Knowledge-free exploration such as mentioned here, uses a fixed probability parameter for randomness. Given a very large search tree, using a fixed exploration factor would make it practically impossible for TOL to choose the most promising node in the frontier, as $(1 - \epsilon)^d$, even for small ϵ, converges to 0 for large tree depth d. Even miniscule amounts of exploration can lead to the most promising node rarely being chosen. This disregard for heuristic exploitation increases exponentially with tree depth. The practicality of exploration might be better suited when a variable parameter is introduced.

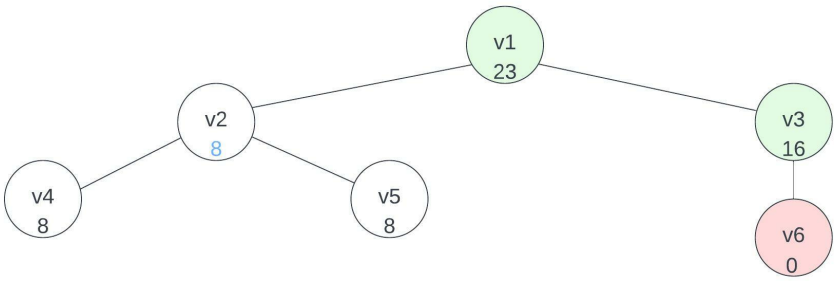

Figure 3.3: TOL-EGBFS with fixed exploration. The green nodes represents nodes which were chosen in the trial, when actions are selected. Red nodes are goal nodes. The best path according to the heuristic would be v2 and then either v4 or v5. Due to exploration, another path is chosen, and in our case leads to a goal.

In figure 3.4 the TOL-EBGFS algorithm is selecting actions to reach an open node. Since in the last iteration the f-value was back propagated to the parent, the left parent is the most promising node. However due to a random action selection, the right node is selected instead and is expanded. The expansion reveals a goal state, that wouldnt have been chosen without exploration. Using an open list with ϵ as the exploration factor, would have a $\epsilon/3$ chance to expand this node, whereas a TOL search has $\epsilon/2$. This difference only increases in magnitude with elevated tree depth.

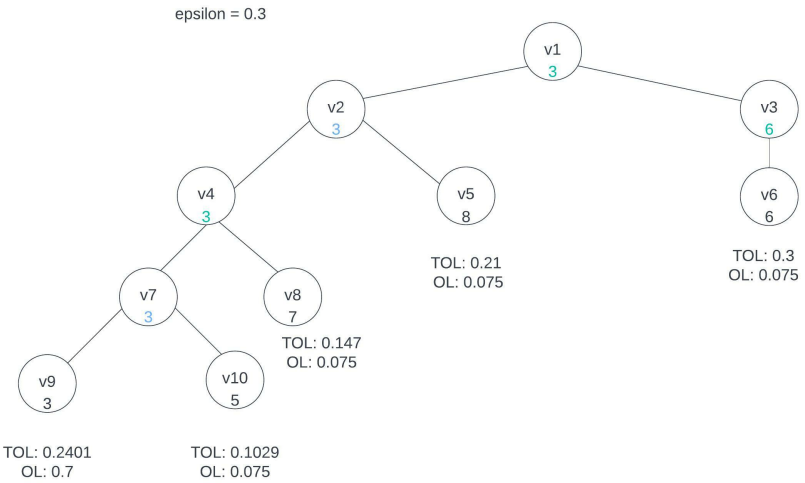

Figure 3.4: Example of choosing a leaf node in an open list vs. trial-based open list. The open list knows nothing of the tree besides having the nodes in the frontier, and so it has only 2 options: choosing the most promising node, v9, or another leaf node at random. For the TOL, the probabilities are much different. Even in a such a shallow tree, the most promising node, which is at depth 4, has a **lower** probability of being chosen than a node in the upper layers, such as v6. A fixed exploration parameter in TOL promotes the generation of the most shallow branches first.

Even though both searches are using the same exploration coefficient, the TOL spreads out the probabilities much more evenly. TOL is more likely to expand v6 as it is to expand the most promising node, v9. The most promising node almost triple as likely to be expanded using explicit open lists than with TOL. The upper layers of the search tree have high chance of exploration with TOL, and it decreases with tree depth.

Algorithm 5 TOL-EGBFS(eps)

```
while not is_timeout() & not is_plan_found() & not out_of_memory() do
    perform_trial()
end while
return plan

procedure perform_trial()
node ← init()
while node.is_closed() do                          ▷ While node is not a leaf node
    node ← select_action(eps)
end while
if s(node) in S* then
    return extract_path(node)
end if
Initialize(node)
backup_queue.add(node)
for n in backup_queue do
    backup(n)
    if not n.equals(init()) then
        backup_queue.add(parent(n))
    end if
end for
```

3.2.2 Knowledgeable exploration

The issue with using a fixed exploration parameter combined with TOL, is that the heuristic exploitation is increasingly ignored with longer tree depth. Knowledge-free exploration comes with the price of not being able to contain itself. The thought of knowledgeable exploration is to bound the exploration in certain ways, so that we reduce its damaging effect on the heuristic exploitation. It is knowledgeable, because it uses search information stored in the nodes. Such information could for example be a visited counter, where each node can tell you how often it has been visited so far. Another example is the current node depth, meaning the path length starting from the root. These variables give us the opportunity to vary exploration, depending on the current node placement, or how often it is chosen. We can do this by creating a function $f : N \mapsto \mathbb{R}_{[0,1]}$, where a node $n \in N$ is given a certain probability of exploration.

Even simple functions, that are easily computed, can be used to vary exploration.

- **Linear Path length** (TOL_{PL1})**:** Keeping track of the longest path in the tree with

depth d_m, and using it to decrease ϵ based on current tree depth d_{node}.

$$\epsilon_{node} = \left(\frac{d_m - d_{node}}{d_m}\right)\epsilon$$

This function linearly reduces exploration with increasing tree depth. The reason for fixed exploration not being that good, is that in deep trees, the most promising leaf node has a miniscule chance of being chosen for expansion. This function attempts to mediate the exploration by removing some of its potency in the deeper layers. A search using a fixed parameter ϵ on a tree of depth d will have a probability of at least $p = (1-\epsilon)^d$, of choosing the most promising leaf node. A search using linear path length increases the lower bound probability to $p = \prod_{i=1}^{d} 1 - \frac{d-i}{d}\epsilon$.
It still promotes exploration in the shallow parts of the tree, in an attempt to start the search off well by exploring the state space first. Then, as the search progresses and the search tree grows in depth, it will exploit the heuristic more. This way, the search is more likely to find many different paths at lower layers. Gradually, however, the search hones in on the goal(s) using the heuristic.

We can extend this logic to create the quadratic path length (TOL_{PL2}), which constrains exploration at a faster rate, going into deeper layers. The function can in fact be generalized to a polynomial path length function (TOL_{PLm}) of degree $m \in \mathbb{N}$:

$$\epsilon_{node} = \left(\frac{d_m - d_{node}}{d_m}\right)^m \epsilon$$

- **Visited count** (TOL_{VC1})**:** Keeping a visited count a for each node can be very helpful in limiting exploration in areas where it is overpowering the exploitation. The idea here is to find the child which paths to the most promising node in the subtree (namely child*) and reduce exploration based on its visit count.

$$\epsilon_{node} = \left(\frac{v_{child*}}{v_{node}}\right)\epsilon$$

Since we are searching through a tree, which is acyclic, the visited counters are ordered by tree hierarchy. A parents visit counter will always upper bound the childs. Meaning if the child has a visited counter equal to the parent, that the child was chosen any time the parent was. This visited counter is good way to quantify the the amount of exploitation vs. exploration happening in a local area of the search tree. If child* has been exploiting the heuristic alot, then the value of epsilon wont change by much. This emphasizes the need for exploration in parts of the search tree in which exploration is scarce. At the same time, it reduces exploration in highly volatile parts of the tree. This function, however, does not necessarily decrease epsilon going down the tree, because the fraction is dependent on only local node values.

- **Geometric avg. of visited count and linear path length** ($TOL_{PL1,VC1}$)**:**

$$\epsilon_{node} = \sqrt{\frac{v_{child*}(d_m - d_{node})}{v_{node} d_m}}\,\epsilon$$

The two functions above focus on two different concerns for limiting exploration. The combination of the two fractions using the geometric average would include both concerns into a single function.

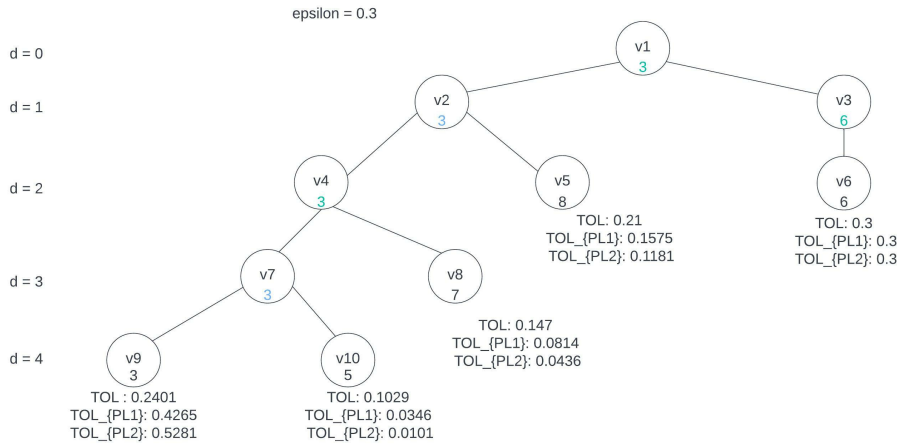

Figure 3.5: Here we showcase the same search tree as above, and the probabilities of the leaf nodes being picked with TOL using linear path length (TOL_{PL1}), quadratic path length TOL_{PL2} and a fixed parameter. TOL with path length (TOL_{PL}) have a higher probability of picked the most promising leaf node, v9. TOL_{PL} is better at exploiting the heuristic in the long run, but in the early stages allow similar exploration as TOL.

Algorithm 6 $TOL_{LP1,VC1}$-EGBFS(eps)

```
while not is_timeout() & not is_plan_found() & not out_of_memory() do
    d_max ← 0
    perform_trial()
end while
return plan

procedure perform_trial()
node ← init()
while node.is_closed() do                          ▷ While node is not a leaf node
    d ← node.get_distance_from_root()
    c ← node.child_with_best_h()                   ▷ can return a list of children
    v_c ← sum_visited_count(c)                     ▷ Sum of visited count of children in c
    node_eps ← sqrt((d_max − d) * v_c/(d_max * node.get_visited_count()))   ▷ local
exploration probability
    node.inc_visited()                             ▷ Increase visited counter
    node ← select_action(node_eps)
end while
if s(node) in S* then
    return extract_path(node)
end if
Initialize(node)
backup_queue.add(node)
d ← node.get_distance_from_root()
if d > d_max then                                  ▷ Keeping d_max updated
    d_max ← d
end if
for n in backup_queue do
    backup(n)
    if not n.equals(init()) then
        backup_queue.add(parent(n))
    end if
end for
```

3.2.3 Monte Carlo methods

Monte Carlo methods[1] address the issue of balancing exploration and exploitation, by choosing the child with the highest expected reward value. It calculates the average reward for all children, and picks the child with the highest average. The reward is a value representing the quality of the node. In game trees, the reward represents the result of the simulation.

Multi Armed Bandit problems (MAB) are solved taking a similar approach to balancing exploration and exploitation. Consider a MAB with n arms, and random variables V_i for $1 \leqslant i \leqslant n$, where V_i represents the i-th arm of the MAB. The goal is to approximate the unknown, true means of the rewards for each arm $\mu_i = E[V_i]$, by considering independent, identically distributed samples. Using a fixed confidence variable $0 < \delta < 1$, we continue sampling the arms until we can choose the arm with the best mean μ_i with probability at least $1 - \delta$.

Median Elimination (ME) is an algorithm for the MAB problem. It chooses a best arm by

iteratively eliminating arms with expected reward $E[V_i]$ smaller than the median of all the arms, until only one arm is left. The algorithm takes as input (ε, δ)-tuple. It outputs an ε-optimal arm with probability at least $(1 - \delta)$[2]. An ε-optimal arm is an arm for which the expected reward is at most ε less than the optimal reward. The elimination of "bad" actions allows the search to exploit more optimal paths. The resulting tree size is reduced, so that search times are generally increased It at the same time reduces search time by not exploring the removed actions. The downside to removing actions from the search tree, is that the successor might later turn out to lead to a goal. Considering the confidence value $\delta = 0.1$, the action removed has a probability of 0.1 of actually having an expected reward above the median value. In large state spaces with many states, this mishap is bound to happen eventually.

Algorithm 7 ME(error,delta) - [2]

$l \leftarrow 1, error_l \leftarrow error/4, delta_l \leftarrow delta/2$
while $|A| > 1$ **do** ▷ While there is more than one action left
  Sample all actions $a \in A$ $\frac{1}{(error_l/2)^2 \log 3/delta_l}$-times
  calculate the average of each action μ_a
  if $\mu_a < Med\{\mu_a | a \in A\}$ **then**
    remove a from A
  end if
  $l \leftarrow l + 1, error_l \leftarrow error * 3/4, delta_l \leftarrow delta/2$
end while
return best action a

There is however no way of coming back once an action is removed. An approach to remedy this, could be to instead save the actions that are removed with ME in a list of forgotten actions. If the prediction of it being a bad action to choose from, is true, we dont care about these forgotten actions. However, in the case of a misprediction, or when none of the "better" actions lead to the goal, we can add them to the children list again.

State space search has a slightly different reward system than game tree search. In a state space search, the reward is the f-value that is back propagated at the end of each trial. The median elimination algorithm can trigger at each inner node, when all children have visited counts above the bound set. It then calculates the median, and removes all actions with an average reward less than the median. However, this may sometimes lead to a conflict between the f-evaluator and ME. In the case that ME tries to remove the best child in terms of the f-value, we will prioritize the evaluator and keep the child.

Algorithm 8 TOL-EGBFS(eps,error,delta) with ME, without forgotten list

while not *is_timeout*() & **not** *is_plan_found*() & **not** *out_of_memory*() **do**
  perform_trial()
end while
return *plan*

procedure *perform_trial*()
node ← *init*()
while *node.is_closed*() **do** ▷ While node is not a leaf node
  node.inc_visited()
  $l \leftarrow node.l()$ ▷ l is the amount of times Median elimination has been triggered on this node before
  $error_l \leftarrow error/4 * (3/4)^l, delta_l \leftarrow delta * (1/2)^l$
  if all children have been visited at least $\frac{1}{(error_l/2)^2 \log 3/delta_l}$-times **then**
    $m \leftarrow$ Median of all childrens average rewards
    for child in node.get_children **do**
      if $child.average\_reward < m$ and child does not have best f-value **then**
        remove child from node.children
      end if
    end for
    node.inc_l() ▷ increment l
  end if
  node ← *select_action*(*eps*)
end while
if *s*(*node*) *in* S^* **then**
  return *extract_path*(*node*)
end if
Initialize(*node*)
backup_queue.add(*node*)
for *n in backup_queue* **do**
  backup(*n*)
  if not *n.equals*(*init*()) **then**
    backup_queue.add(*parent*(*n*))
  end if
end for0

4

Experiments

4.1 Fast Downward

Fast-Downward is a free and open-source, redistributable project, which was founded in 2003 by Malte Helmert [3] & Silvia Richter. A few years later, Fast-Downward merged with LAMA, which was founded by Silvia Richter & Matthias Westphal based on the original FD.

FD eventually merged with FD-Tech in 2011, which was made by Erez Karpas & Michael Katz. This formed the system which is currently known as Fast Downward today.

The goal of the FD project was to build an independent, classical AI planning system. Fast-Downward uses many different evaluators and heuristics to estimate distance from goal state and to determine which states to expand next. Fast-Downward has a choice of different search algorithms as well, which help us find a path toward the goal. There are many search options available, such as setting time or memory limits, choosing between tree or open-list search-engines, reopening closed nodes, or setting preferred operators.

FD is constantly being updated, and has remained lightweight and fast over the years, becoming one of the most recognized and consistent planning systems contributing to the AI planning community[1],with many of the top planners based on the FD codebase.

[1] https://planning.wiki/ref/planners/fd

4.2 Experimentation and comparison of trial-based open list (TOL) and explicit open list (OL)

We now run the *default optimal suite*, comprised of 64 domains with 1797 tasks in total. However, there are certain domains with very costly plans, and so the relative differences of these tasks are much higher. Other tasks might be overlooked in terms of cost, so instead there are two batch jobs, where high-cost (over 100'000 per task) and the lower cost ones are seperated. In the costly table we have the domains *parcprinter-08-strips* and *parcprinter-opt11-strips*, all the other 62 domains are in the lower-cost table. Since the costly batch consists of few tasks, any statistical analysis on it will be less precise than on the cheap batch, which is larger in number of tasks by a factor of 35. We will therefore be using the cheap task batch for analysis. We use the h^{FF} heuristic, as it has a very fast translation time. The FF-heuristic is a heuristic than can very quickly compute the heuristic values of a state space, and this will leave more time for the search to complete.

4.2.1 (OL)-EGBFS vs TOL-EGBFS

Cheap tasks with reopening (1747)									
Results	OL-0.3	OL-0.2	OL-0.1	OL-0.05	TOL-0.3	TOL-0.2	TOL-0.1	TOL-0.05	TOL-0.01
Coverage	1506	1498	1494	1494	1137	1306	1444	1505	1548
Cost	128504	128115	127660	129783	105202	111300	119614	120550	131376
Memory	35229036	34645528	36120352	38496812	136844072	46611520	35180276	33295648	36588732
Search Time	0.15	0.15	0.14	0.14	1.76	0.62	0.26	0.17	0.19

Cheap tasks without reopening (1747)									
Results	OL-0.3	OL-0.2	OL-0.1	OL-0.05	TOL-0.3	TOL-0.2	TOL-0.1	TOL-0.05	TOL-0.01
Coverage	1506	1505	1497	1492	1134	1294	1454	1501	1548
Cost	127773	128719	131403	129850	109667	115328	120750	122749	131391
Memory	34121888	34483204	38025116	37775024	158902144	50886072	36366868	34254224	36970164
Search Time	0.15	0.14	0.14	0.14	1.89	0.67	0.26	0.18	0.19

OL-EGBFS seems to have a relatively consistent coverage when varying ϵ. This can be explained by the way the epsilon greedy open list from Valenzano et al.[5] chooses a random successor. The min heap used for the open list is implemented with a vector. This vector is maintained is such a way, that the minimum value is the first element. Consider a node element with placement i in the vector. The children of this node in the min heap, correspond to the elements with placements $2i, 2i + 1$ in the vector. Since a min heap is a binary tree, this is well-defined behaviour. The random selection from the vector is therefor equivalent to a random selection from a min heap structure, and our arguments from chapter 3 still hold. Since OL-EGBFS can choose a random successor without adding complexity, the search times and coverage are very similar. For TOL, however there is a decrease in coverage, search time and memory with decreasing ϵ. The smaller ϵ-value leads to exponentially less exploration, and as a result, evaluations. In evaluating less states, we find increasingly expensive solutions. Since the searches for TOL without reopening are

worse in every aspect, we will continue comparisons using only searches *with* reopening.

4.2.2 TOL-EGBFS vs TOL$_{PL2,VC2}$-EGBFS (PV2)

Cheap tasks with reopening (1747)									
Results	TOL-0.2	TOL-0.1	TOL-0.05	TOL-0.01	PV2-0.3	PV2-0.2	PV2-0.1	PV2-0.05	PV2-0.01
Coverage	1309	1444	1504	1548	1325	1407	1477	1510	1544
Cost	111524	119798	120797	131521	113474	115080	120943	122721	129729
Memory	46482632	35343332	33600284	36634580	47783252	39184080	36043580	34637792	37309104
Search Time	0.75	0.32	0.22	0.19	0.79	0.45	0.29	0.22	0.19

When comparing PV2 to TOL, the general increase in quality with decreasing ϵ is still visible, but less extreme. PV2-0.3 vs TOL-0.3 shows an increase of almost 200 coverage on the optimal suite. However, the relative coverage improvements also decrease for $\epsilon \to 0$, as for 0.01, PV2 lags behind slightly in coverage to its TOL counterpart.

4.2.3 TOL-EGBFS with ME

Cheap tasks with reopening and exploration $\epsilon = 0.2$ (1747)								
Results	(0.2-0.1)	(0.2-0.05)	(0.4-0.1)	(0.4-0.05)	(0.6-0.1)	(0.6-0.05)	(0.8-0.1)	(0.8-0.05)
Coverage	1372	1376	1393	1387	1394	1396	1401	1398
Cost	159025	158404	163060	165543	165862	166122	167251	173801
Memory	108452204	105793064	67260888	68005452	60134664	56523064	55252500	55069228
Search Time	1.65	1.61	1.28	1.28	1.11	1.08	0.97	0.93

Cheap tasks with reopening and exploration $\epsilon = 0.3$ (1747)								
Results	(0.2-0.1)	(0.2-0.05)	(0.4-0.1)	(0.4-0.05)	(0.6-0.1)	(0.6-0.05)	(0.8-0.1)	(0.8-0.05)
Coverage	1335	1343	1361	1363	1362	1362	1363	1370
Cost	152371	154835	156696	155729	161729	161864	167232	164805
Memory	175805344	160400692	90151524	87744812	73113596	71566088	65689780	62464568
Search Time	3.10	2.96	2.11	2.04	1.73	1.64	1.40	1.33

Already just looking at TOL with ME, we can see the difference to the results for TOL and OL from above. The coverages are lower. This might be due to removing children, only to add them again later, when all the other options have been considered. Although, eventually TOL with ME should find the same solutions as TOL, the removal of children slows the process down. For low ϵ, TOL with ME seems to speed up, in line with the other TOL searches. For the $(error, delta)$ parameters, coverage quality favors higher error acceptance and a relatively low confidence value.

4.2.4 Comparison

Having used the default optimal suite to find the quickest versions of each search, we can now move on to the *default satisficing suite*. It consists of 2742 tasks in 83 domains. Again, as in the last section, we will remove very expensive tasks, as they influence the cost too much for useful assessment. We will remove 2 domains, namely *parcprinter-08-strips* and *parcprinter-sat11-strips*. This leaves us with 81 domains and 2692 tasks.

Satisficing track cheap tasks, with reopening (2692)								
Results	ME(0.2,0.8,0.05)	ME(0.2,0.8,0.1)	PV2-0.05	PV2-0.01	TOL-0.05	TOL-0.01	OL-0.3	OL-0.4
Coverage	1664	1665	1897	1964	1841	1997	1878	1851
Cost	88642	88199	91038	94722	90743	94361	99130	97049
Memory	105209140	100244792	49293224	47105748	53639352	44974748	46687472	48394208
Search Time	1.72	1.68	0.64	0.47	0.76	0.46	0.54	0.59

The combined experiment on the satisficing track bring the quickest searches together. In very last place, with more than double the memory usage of fellow searches, we have TOL with ME. The coverage is low, at 1665, and the small increase in cost doesnt make up for the big difference in search time.

The OL searches have the worst cost of all searches, but the coverages are much better than TOL with ME. The memory used is also lower, which puts another point in how inefficient TOL with ME actually is. The OL searches are however worse in all aspects but memory, comparing to the top 2 competitors PV2 and TOL.

That being said, TOL-0.01 takes the spotlight. Exploration in a TOL search exponentially decreases for linear decrease in ϵ, and has given better results even, than using a varying factor.

Comparing PV2-0.05 and TOL-0.05, we can see that PV increases coverage up until a certain lower bound $0.01 < b < 0.05$, as with memory and search time.

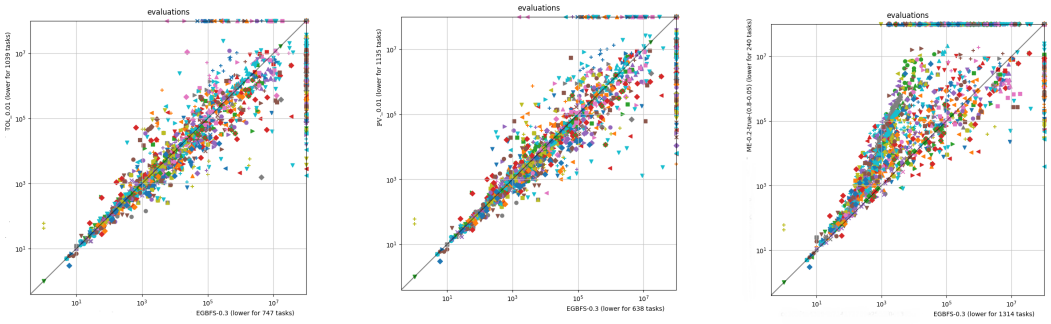

Figure 4.1: Figures showing evaluations for OL-0.3 vs TOL-0.01 (left), OL-0.3 vs PV2-0.01 (middle) and OL-0.3 vs ME-0.2-0.8-0.05. Both TOL and PV2 seem to grow linearly in evaluations. TOL with ME and $\epsilon = 0.2$, on the other hand has a line forming the upper bound. The upper bound seems to be polynomial, around x^2. TOL with ME search has a lower coverage, because it evaluates too many states, from having too much exploration.

5
Conclusion

In conclusion, the TOL search as first mentioned in Schulte and Keller[4] by the name THTS can be very advantageous for exploration. Splitting the task of producing a frontier node up into indivual action selection allows for a variety of methods to be used, that is infeasible for OL search. The action selection can be modeled as a multi-armed bandit problem for solutions stemming from statistical decision theory, such as median elimination. Although median elimination was showcased to not be efficient, the same might not be said from other methods. Even-Dar[2] mentions *successive elimination* as an alternative PAC-Bound (Probably Approximately Correct) algorithm, which might be more effective. Even so, other, simpler methods for balancing exploration in TOL are shown to have an impact. $\text{TOL}_{PL2,VC2}$ uses a geometric average of quadratic path length and quadratic visited count, to reduce exploration more than that of a regular TOL search. TOL with exploration parameters all have in common, that the parameter should remain relatively small. Exploration explodes in deeper search trees. The best-f leaf node at a depth 100 will be picked with the low probability of 36.6% for a fixed exploration parameter $\epsilon = 0.01$. While decreasing the exploration in deeper layers of the tree can remedy this issue, using a function declining in depth such as polynomial path length (TOL_{PLn}) brings in new problems. The non-best leaf nodes in deep layers have almost no chance of being expanded, by far less than in TOL, and infinitesimal compared with OL. Additionally, polynomial path length provides negligible support for exploitation in the upper layers of the tree. The search might be better off with exploration *increasing* in deeper layers instead. Changing polynomial path length, so that exploration increases in deeper layers can perhaps fix or lessen the issues, while still keeping best-f nodes the priority.

Although TOL-0.01 came through on top in our experiments, there is ample chance for there to exist methods to balance exploration in more consistent ways. But although Monte Carlo methods and PAC-algorithms balance exploration in their own right, the inner proceedings differ to $\epsilon-$Exploration. With MC and PAC, there is no randomness in the sense of choosing an arbitrary action, but there is a certain amount of doubt that the action was not up to standards. With $\epsilon-$Exploration, we doubt an already given estimate of the best action, and with some probability we take another one instead. The differences make it

difficult to find a seamless combination of the two procedures, in which neither party comes in conflict with the other.

TOL is a powerful way to elaborate the selection process of a frontier node. It allows a refined process of selection, in which exploration can be manipulated in minute detail. However the chaining of such selections also reveals its vulnerability, as it gradually hinders the exploitation of the heuristic.

Bibliography

[1] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *Journal of Artificial Intelligence Research (2006)*, 4(1):1—49, 2012.

[2] Eyal Even-Dar, Shie Mannor, and Yishay Mansour. Action elimination and stopping conditions for the multi-armed bandit and reinforcement learning problems. *Journal of Machine Learning Research*, 7:1079–1105, 2006.

[3] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[4] Tim Schulte and Thomas Keller. Balancing exploration and exploitation in classical planning. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search (SoCS 2014)*, pages 139–147, 2014.

[5] Richard Valenzano, Nathan R. Sturtevant, Jonathan Schaeffer, and Fan Xie. A comparison of knowledge-based gbfs enhancements and knowledge-free exploration. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS2014)*, pages 375–379, 2014.

University of Basel

Faculty of Science

Declaration on Scientific Integrity

(including a Declaration on Plagiarism and Fraud)
Translation from German original

Title of Thesis:

Name Assesor: Malte Helmert

Name Student: Rasmus Jensen

Matriculation No.: 17-947-664

With my signature I declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Place, Date: Basel, 15.07.2022 Student: Rasmus Jensen

Will this work be published?

☐ No

☒ Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: 15.07 or whenever possible

Place, Date: Basel, 15.07.2022 Student: Rasmus Jensen

Place, Date: _____ Assessor: _____

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis .