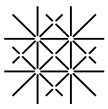

University of Basel

Diversifying Greedy Best-First Search by Clustering States

Bachelor Thesis

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

Examiner: Malte Helmert
Supervisor: Jendrik Seipp

Daniel Killenberger
daniel.killenberger@unibas.ch
2013-059-647

01.03.2017

Acknowledgments

I would like to thank Jendrik Seipp for all the helpful advice during and outside of our weekly meetings. I would also like to thank Prof. Dr. Helmert for inspiring me to pursue this thesis in his group with the Lecture 'Foundations of Artificial Intelligence' and giving me the possibility to do so. Thanks also to Gordon Mickel, Matthias Chinyen Tsai and Elliot Walmsley for proof reading. Calculations were performed at sciCORE (http://scicore.unibas.ch/) scientific computing core facility at University of Basel.

Abstract

Greedy best-first search has proven to be a very efficient approach to satisficing planning but can potentially lose some of its effectiveness due to the used heuristic function misleading it to a local minimum or plateau. This is where exploration with additional open lists comes in, to assist greedy best-first search with solving satisficing planning tasks more effectively. Building on the idea of exploration by clustering similar states together as described by Xie et al. [2014], where states are clustered according to heuristic values, we propose in this paper to instead cluster states based on the *hamming distance* of the binary representation of states [Hamming, 1950]. The resulting open list maintains k buckets and inserts each given state into the bucket with the smallest average hamming distance between the already clustered states and the new state. Additionally, our open list is capable of reclustering all states periodically with the use of the k-means algorithm.
We were able to achieve promising results concerning the amount of expansions necessary to reach a goal state, despite not achieving a higher coverage than fully random exploration due to slow performance. This was caused by the amount of calculations required to identify the most fitting cluster when inserting a new state.

Table of Contents

1 Introduction

Classical planning is an approach to automated problem solving, which assumes a simplified environment for an agent to work within. Such an environment is defined by its initial state, a set of operators that can be applied by the agent to modify the environments state and a goal, which the agent is supposed to reach.
Greedy best-first search (GBFS) is a technique used in classical planning that tries to find a solution by expanding states evaluated as being closest to a goal state according to a heuristic function. A so called *open list* usually makes the decision which state is going to be expanded next by the search algorithm. In the case of GBFS the open list is implemented as a priority queue which stores inserted states sorted by their heuristic values. This has proven to be an effective method to solving satisficing planning tasks in many domains, without any guarantee for optimality. Due to being highly dependent on the employed heuristic function, GBFS has the potential to fail in some domains due to the heuristic being misleading, which can lead the search to a local minimum or plateau [Xie et al., 2014]. By *diversifying* the state space of GBFS with an additional open list to allow for the exploration of a wider selection of states, significant performance gains can be achieved as shown by Xie et al. [2014]. The mentioned paper describes the implementation of a *Type-based* open list that clusters states based on their *type*. The type of a state can be defined arbitrarily but is usually a combination of the state's heuristic value for different heuristics and it total path cost. These type attributes act as a key to identify which cluster a state is being assigned to. If no cluster exists that can be associated with a state's key, the Type-based open list creates a new cluster. When asked to return a state, the open list returns a random state from a random cluster.
In this paper we build on the idea of diversifying GBFS with an open list that clusters similar states, but define the similarity of states differently. We evaluate states as more dissimilar the greater the *distance* between them is. We define said distance to be the *hamming distance* between their respective binary representations [Hamming, 1950]. When inserting a new state s into what we call the *Cluster-based* open list, s is assigned to one of k clusters with the minimal average distance between the states contained within the cluster and s. When the search algorithm asks for a new node to expand, the Cluster-based open list returns a random state from a randomly chosen cluster. Additionally we provide the option to periodically reassign all states to more fitting clusters using the k-means algorithm to correct for sub optimal assignments. It is possible that the suggested way of clustering states proves to be a better way of distinguishing between dissimilar states than the type of a state. If this holds true and every cluster contains only very similar states we expect to only have to look at few of the states within a cluster to introduce fruitful exploration and consequently reduce the amount of expansions required to solve a given planning task.
When evaluating our experiments, we noticed a trend of reducing the amount of required expansions for commonly solved hard planning tasks when using Cluster-based exploration compared to random or Type-based exploration. This trend is even more apparent when compared to GBFS without exploration. Due to the implementation being not as efficient as we desired it to be, we weren't able to compete with the coverage of random or Type-based exploration or even GBFS alone.
This paper is organized as follows; first we introduce the reader to the theoretical background of classical plan-

ning, the SAS^+ formalism, and planning within state spaces. We then provide some background on open lists and heuristics in order to then briefly explain GBFS. We will then elaborate on why the diversification of GBFS with an additional open list can make it more effective at solving satisficing planning tasks. We follow up with the description of the two proposed algorithms for the Cluster-based open list (one with and one without k-means reclustering) and end with the evaluation of the conducted experiments.
The implementation has been realized within the *Fast Downward* planning system [Helmert, 2006].

2

Theoretical Background

2.1 Classical Planning

> "Planning is the reasoning side of acting. It is an abstract, explicit deliberation process that chooses and organizes actions by anticipating their expected outcomes." [Ghallab et al., 2004, p. 1]

A classical planning task consists of an initial state, a goal and a set of actions that determine how the state of the environment can be changed [Weld, 1999, p. 93]. We formally define a classical planning task as follows:

Definition 1. Given $\Sigma = (S, s_0, s^*, \mathcal{O})$ where S is the set of possible states, s_0 the initial state, s^* the goal and $\mathcal{O}$ a set of operators, find a sequence of operators $\langle o_1, o_2, ..., o_k \rangle$ such that $s_0 \xrightarrow{o_1} s_1 \xrightarrow{o_2} s_2 \dots s_{k-1} \xrightarrow{o_k} s_k$ and s_k fulfills s^*, or prove that no such sequence exists [Ghallab et al., 2004, p. 10].

Classical planning is one of many strategies to automated problem solving that assumes the system to be finite, fully observable, static and all operators to be deterministic [Ghallab et al., 2004, p. 9-10].
In order to find *optimal* solutions to classical planning problems, one can introduce a cost function $cost : \mathcal{O} \to \mathbb{R}_0^+$ that assigns a non-negative cost to every operator. The optimal solution is then a path $\pi = \langle o_1, o_2, ..., o_k \rangle$ where $\sum_{i=1}^{k} cost(o_i)$ is minimal. Since optimal solutions are not the goal of satisficing planning, costs will be ignored in this work.

2.2 SAS+ Formalism

There are multiple ways of describing a classical planning task. The one we will be focusing on is SAS^+ as it is the formalism used internally in the *Fast Downward* planning system. The SAS^+ formalism uses *multi-valued state variables* to describe states. Each of the state variables has a *domain*, that defines a finite set of valid values. Operators have a *precondition* and an *effect*. The *precondition* specifies which variables must have which values before the operator can be executed. The *effect* defines which variables will be changed to which values after the operator has been applied. We formally define a SAS^+ planning problem as follows [Bäckström and Nebel, 1995].

Definition 2. A SAS^+ planning task is given by $\Pi = \langle V, \mathcal{O}, s_0, s^* \rangle$ where V is a finite set of *variables*, $\mathcal{O}$ is a finite set of operators, s_0 is the initial state and s^* the goal condition. Each variable $v \in V$ has an associated finite *domain* D_v. We define a *fact* as a pair $\langle v, d \rangle$ with $v \in V$ and $d \in D_v$ and a set of facts as *consistent*, if all facts belong to different variables. We define a *partial assignment* p as a consistent set of facts. We denote the set of variables contained in p as $vars(p)$. If $vars(p) = V$, p is called a *state*. We denote the value of a variable v in state s as $s[v]$. We will refer to the set of all states as S_V and to the set of all partial assignments as P_V. We define $o \in \mathcal{O}$ as a pair $\langle$pre, eff$\rangle$, where *pre, post* $\in P_V$ define the *precondition* and *effect*. The initial state s_0 is a state and therefore $s_0 \in S_V$. The goal condition is similar to a precondition in that it is a partial assignment and therefore $s^* \in P_V$.

It is easy to see that every finite domain D_v associated with variable $v \in V$ can be projected on to $\{0, 1, \ldots, |D_v| - 1\} \subset \mathbb{N}_0^+$ using a bijective function $f : D_v \to \{0, 1, \ldots, |D_v| - 1\}$. This is important to note as we can then standardize all variables $v \in V$ to have an associated domain $D_v = \{0, 1 \ldots, |D_v| - 1\}$. This standardized form is how the *Fast Downward* planning system [Helmert, 2006] describes state variables and their domains and we assume all domains to be standardized in this way going forward.
Tasks described by a formalism other than SAS^+ e.g. *PDDL* or *STRIPS* are being translated without any loss of information by the *Fast Downward* planning system before the initialization of the actual search.

2.3 State Space Planning

A planning task $\Pi = \langle V, \mathcal{O}, s_0, s^* \rangle$ induces a state space consisting of nodes that correspond to states $s \in S_V$, edges that correspond to operators $o \in \mathcal{O}$ and the set of goal nodes that correspond to the set of states $S^* \in S_V$ that fulfill the goal condition $s^* \in P_V$. The set of all states is initially not explicitly given but instead continually generated by applying operators to existing states. The state resulting from the effect of an operator applied to a state s is called the child of s [Ghallab et al., 2004, p. 69; 544]. Generating all children of a particular state s is called *expanding* s. The plan for solving the given planning task corresponds to the path within the induced state space that leads from the root node (initial state) to a node associated with a goal state.

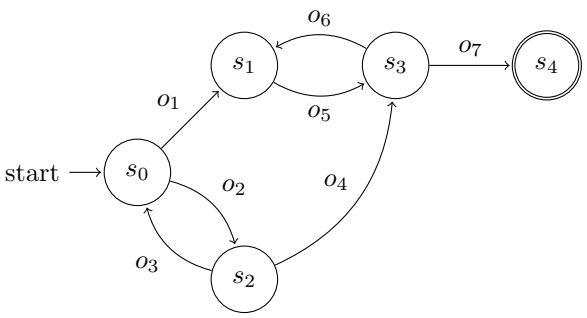

Figure 2.1: Example of an explicitly generated state space where $s_4 \in S^*$

.

In order to recreate the path that solves the given planning task and to calculate the final cost, nodes can not just be representative of states. Instead they have to be implemented as data structures which we'll call *search nodes* that contain the following information: [Russel et al., 1995, p. 72]:

- the state the node corresponds to
- the node that generated this node (parent node)
- the operator that was applied to get from the parent node to this node
- the number of nodes on the path to this node (depth of the node)
- the path cost from the initial state to this node (not important in our case)

As we expand nodes we need another data structure to be able to store the generated nodes for later expansion. This is where *open lists* come in to play.

2.4 Open Lists

Open lists are data structures used to temporarily store search nodes that have been generated by expanding a parent node. Stored nodes can later be retrieved by the search algorithm for their expansion. An open list must implement three basic functions to work with a state space search algorithm.

- open.is_empty() returns true if no nodes are stored in the open list, else return false

- open.pop() removes and returns the next node to expand
- open.insert() inserts the node into the open list

These functions can be implemented differently depending on the requirements of the open list.

2.5 Heuristics

> "Heuristics are criteria, methods, or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal. They represent compromises between two requirements: the need to make such criteria simple and, at the same time, the desire to see them discriminate correctly between good and bad choices." [Pearl, 1984, p. 3]

Applied to state space search, heuristics are usually functions $h : S \to \mathbb{R}_0^+$, where S is the set of states of the given planning task, that assign a real value to every state to approximate the state's distance to the task's goal. If $h(s)$ were a perfect heuristic it would return the exact cost required to reach the nearest goal state, return $h(s) = 0$ if $s \in S^*$ and $h(s) = \infty$ if no solution can be found originating from s.

2.6 Greedy Best-First Search

Greedy Best-first Search(GBFS) is an informed search algorithm that uses a heuristic function to decide which states to expand next while ignoring the cost of operators. We formulate Algorithm 1 to describe GBFS using an open list. Said open list is implemented as a priority queue sorted by the heuristic value of the nodes in ascending order. Algorithm 1 assumes states $s \in S$ to be contained in search nodes n which we require to keep track of the path. We use the function *get_path*(n) to recursively build the path that leads from n_0 (the initial node) to n. Additionally we assume to have a function *expand*(n) that returns the set of all nodes $\mathcal{N}'$ reachable through the effect of operators that are applicable to s contained in n.

Algorithm 1 Greedy Best-First Search

```
n ← n_0
open ← empty list
while n.s ∉ S* do
    N' ← expand(n)
    for n' ∈ N' do
        open.insert(n')
    if open.is_empty() then
        return failure
    n ← open.pop()
return get_path(n)
```

Using solely heuristics to decide how to move towards a goal within a given state space is a very useful tool in many domains, but can result in mistakes, caused by sibling nodes being expanded in the wrong order. The reason for this is that the root node has a lower heuristic value than its siblings even though its subtree contains very hard to find solutions, or no solutions at all. To lessen the impact of such mistakes we can *diversify* GBFS by exploring a wider range of states [Xie et al., 2014, p. 2]. We will now discuss how this can be realized.

2.7 Diversification of Open Lists

Instead of just using one open list, e.g., an open list sorted by the value of a heuristic function, it proved very profitable to diversify the search by introducing a second or more open lists dedicated to exploration. This makes the search algorithm look at states that the heuristic function would normally ignore, which often results in finding a goal state more efficiently as shown by Xie et al. [2014].

The *Fast Downward* planning system implements an alternating open list, that allows the search algorithm to use multiple open lists. For this purpose the alternating open list inserts all generated nodes into all underlying open lists. When the search algorithm ask for a node it chooses which open list to pop the node from in a *round robin* manner.
Xie et al. [2014] implemented an exploration open list which clusters states based on their *type* which is defined by multiple heuristics and or its total path cost as type attributes, putting the states with equal values for all attributes into the same cluster and generating a new cluster if no cluster with such type-attributes exists. Popping from the open list returns a random node from a random cluster. This lead to a significant performance boost in many domains. Their work forms the basis for the idea of this bachelor thesis concerning the clustering of states with similar attributes. We will explain next how the definition of similarity and the creation of clusters in our proposed algorithms differs from the approach by Xie et al. [2014].

3
Cluster-based Open List

The main objective of this bachelor thesis is to develop an open list that clusters states based on their *similarity* and pops random states from randomly chosen clusters. This should help diversify the search when using greedy best-first search and therefore avoid early mistakes caused by a misleading heuristic, when both open lists are used in conjunction. One benefit is that clustering by similarity may prove to be faster than evaluating heuristics as done by the Type-based open list. Additionally it may prove that, by clustering based on similarity, and therefore reducing entropy within clusters, exploration becomes more effective which would lead to reducing the amount of required expansions to reach a goal state.

Using a tuple notation for states, intuitively $\langle 3, 3, 2\rangle$ is a lot more similar to $\langle 3, 3, 3\rangle$ than to $\langle 2, 1, 0\rangle$. With this idea in mind we need a formal way for describing similarity. For this purpose we transform states to a binary representation: for variable $v \in V$ in state s the associated domain D_v is of standardized form which indicates $s[v] \in \{0, \ldots, |D_v| - 1\}$. We represent the state with a binary string where for each variable there are $(|D_v| - 1)$ $0's$ and a 1 at index $s[v]$. Chaining all resulting binary strings, which correspond to the states' variables, we get the binary representation of the given state. We call this binary representation *bit mask* and it is created according to Algorithm 2. Said bit mask will be stored as a vector of booleans. We will call the contained bits *facts* and they can be addressed by the the index that corresponds to their position within the bit mask: $bit\_mask[i]$. The total number of facts N is calculated according to Equation 3.1.

$$N = \sum_{v \in V} |D_v| \tag{3.1}$$

Algorithm 2 Calculate Binary Representation of States

```
function BIN(s)
    bit_mask ← new vector of size N
    i ← 0
    for v ∈ V do
        bit_mask[i + s[v]] ← True
        i ← i + |D_v|
    return bit_mask
```

Let us go through an example where state s is defined by a set of two variables $V = \{v, w\}$ with $|D_v| = 3$, $|D_w| = 2$, $s[v] = 0$ and $s[w] = 1$. The following vector would then be the binary representation of state s according to Algorithm 2: $bin(s) = \langle 1, 0, 0, 0, 1\rangle$, where the first three bits of the binary representation represent variable v and the last two bits represent variable w.

Now, to calculate the similarity or rather the opposite, the *hamming distance* between two states, one only needs to sum the number of *facts* that are different [Hamming, 1950].

Definition 3. We define the *distance* d between two states s_1 and s_2 as:

$$d(s_1, s_2) = \sum_{i=1}^{N} e(bin(s_1)[i], bin(s_2)[i]) \tag{3.2}$$

where N is the total number of facts and

$$e(x, y) = \begin{cases} 0, & \text{if } x = y \\ 1, & \text{otherwise} \end{cases} \tag{3.3}$$

Now the goal is to cluster states together based on which states are closest to each other. For this purpose we want to be able to calculate the distance between not only states but also the *average* distance between a state and all states within a cluster. In order to more efficiently calculate the distance between a state and a cluster of states, we save a vector per cluster that saves how many states' binary representation hold a specific fact true ($factsCount$) and also the total amount of states stored in a given cluster ($numStates$). For example: Cluster c already stores s_1 with $bin(s_1) = \langle 1, 0, 0, 0, 1 \rangle$ therefore $c.factsCount = \langle 1, 0, 0, 0, 1 \rangle$ and $c.numStates = 1$. If we were to insert s_2 with $bin(s_2) = \langle 1, 0, 0, 1, 0 \rangle$ into c then $c.factsCount = \langle 2, 0, 0, 1, 1 \rangle$ and $c.numStates = 2$.

Definition 4. We define the *average distance* d between a state s and a cluster of states c as:

$$d(s, c) = \sum_{i=1}^{N} \left| \frac{c.factsCount[i]}{c.numStates} - bin(s)[i] \right| \tag{3.4}$$

where N is the total number of facts.
For an additional speed boost we introduce an additional vector associated with every cluster, that stores the mean value for every fact ($factsMean$) and has to be recalculated every time a state is being inserted into or popped from a cluster:

$$c.factsMean[i] = \frac{c.factsCount[i]}{c.numStates} \tag{3.5}$$

which means we can simplify Equation 3.4 to:

$$d(s, c) = \sum_{i=1}^{N} |factsMean[i] - bin(s)[i]| \tag{3.6}$$

Compared to the Type-based open list there is no easy way to decide on the amount of clusters. This means that the open list will be using a static amount k which can be altered by a parameter. The open list creates the bit mask at the point of inserting a newly generated node and replaces state s to be the newly generated bit mask instead, as we have no use for the original state representation in the Cluster-based open list. It inserts nodes into an empty cluster until every cluster contains at least one node. Afterwards if there are no empty clusters left, the open list calculates the distance between the state, associated with the node that is to be inserted, and all clusters. It then inserts the node into the cluster that has the smallest distance to the state contained within the node according to Equation 3.6. To pop a node it chooses a random node from a random cluster which then gets removed and returned. We assume variables $factsCount$, $numStates$ and $factsMean$ to automatically be adjusted correctly. This results in Algorithm 3.

Algorithm 3 Cluster-based open list

function CALCULATE DISTANCE(c, b)
  $d \leftarrow 0$
  $i \leftarrow 0$
  for $mean \in c.factsMean$ **do**
    $d \leftarrow distance + abs(b[i] - mean)$
    $i \leftarrow i + 1$
  return d

function DO INSERTION(n)
  $n.b \leftarrow bin(n.s)$
  delete $n.s$
  if there are empty clusters **then**
    insert n into an empty cluster
  else
    insert n into cluster c with minimal distance d =CALCULATE_DISTANCE($c, n.b$).

function POP
  return and **remove** random node from random nonempty cluster

function EMPTY
  return $numStates == 0$

4

Cluster-based Open List with K-Means Reclustering

In order to correct for badly clustered states, especially at the start of Algorithm 3 where nodes are just filling up empty clusters, we implemented Algorithm 4 to reassign nodes using k-means clustering. It is very fitting to use k-means considering us already having k clusters with calculated mean vectors ($factsMean$) to represent them. Standard k-means clustering is composed of four steps:

1. Initialize: Generate k random mean vectors.
2. Cluster: Associate every data point with the closest mean.
3. Centroid: Calculate the centroid of every cluster to get the new representative mean vectors.
4. Convergence: Check for convergence. If criteria not yet fulfilled jump to step 2.

As we already have calculated mean vectors we will skip step 1 and we will replace the stop criterion to check for a given time limit instead of checking for convergence. As it is difficult to decide when to reassign nodes, the open list will do so periodically (time interval) and then cluster for the given time limit. The open list will check if the time interval has been reached at the end of the insertion method. It is important to note that k-means may produce empty clusters. The resulting algorithm is described in Algorithm 4. For the corresponding C++ implementation see Appendix A.

Algorithm 4 Cluster based open list with K-Means Reclustering

```
function CALCULATE DISTANCE(c, b)
    d ← 0
    i ← 0
    for mean ∈ c.factsMean do
        d ← distance + abs(b[i] − mean)
        i ← i + 1
    return d

function DO INSERTION(n)
    n.b ← bin(n.s)
    delete n.s
    if there are empty clusters then
        insert n into an empty cluster
    else
        insert n into cluster c with minimal distance d =CALCULATE_DISTANCE(c, n.b).
        insert n into the set of all stored nodes 𝒩
    if time interval has been reached then
        K-MEANS()

function POP
    return and remove random node from random nonempty cluster

function EMPTY
    return numStates == 0

function K-MEANS
    while time limit has not been reached do
        for n ∈ 𝒩 do
            assign n to cluster with minimal distance d =CALCULATE_DISTANCE(c, n.s).
        recalculate means of clusters
```

5 Evaluation

In this chapter we will elaborate on how well the two implementations described in Chapter 3 and Chapter 4 solved common satisficing planning tasks when used in conjunction with GBFS. As described in Section 2.7 and Chapter 3 the goal was to improve GBFS by diversifying the search through the exploration of states that are dissimilar to each other. We will be looking at how well the Cluster-based open list is able to improve GBFS, which is using the FF-heuristic [Hoffman and Nebel, 2001] as its heuristic evaluator function, to a higher solving rate or *coverage* of common satisficing problems. Furthermore we will look at how well the Cluster-based open list was able to reduce the amount of required expansions. The FF-heuristic has been established as *the* heuristic to compare new algorithms to as it solves many planning tasks very efficiently. As shown by Valenzano et al. [2014] greedy best-first search can be enhanced with random exploration to reach higher coverage. We confirm this in Table 5.2. This means we will try to improve upon the baseline of GBFS using FF combined with the Cluster-based open list using only one cluster for exploration, which is equal to random exploration. In our experiments we will be ignoring the cost of operators as it leads to a higher coverage using the FF-heuristic as shown by the LAMA planner [Richter and Westphal, 2010, p. 155].
In our series of experiments we first compare Cluster-based exploration to random exploration for different amount of clusters. We go on to compare different reclustering parameters in order to determine which ones to use for the following comparisons of reclustering vs no reclustering, Cluster-based exploration (with and without reclustering) vs Single FF, and Cluster-based exploration (with and without reclustering) vs Type-based exploration.
The experiments have been conducted using the *lab*[1] Python module and computed on *sciCORE* [2] with a three minute time limit and a memory cap of two gigabytes.
From here on we will always assume that the compared implementations of exploration open lists are being used in an alternating open list in conjunction with GBFS that uses the FF-heuristic, unless explicitly stated otherwise.
The search options that have been used for the *Fast Downward* planning system can be found in Appendix B.

5.1 Comparing the Amount of Clusters

The following experiments have been conducted to determine the amount of clusters k that lead to a better performance than just one cluster, which is equal to random exploration. We will be comparing $k = \{1, 10, 100, 1000, 10000, 100000\}$ with each other over 2532 satisficing planning tasks using the algorithm described in Chapter 3.

As it turns out we didn't manage to get a higher coverage over all tasks for any $k > 1$ as shown in Table 5.1. Just a few domains i.e. *mistery*, *nomisteray-sat11-strips* and *thougtful-sat14-strips* were solved more effectively

[1] http://lab.readthedocs.io/
[2] https://scicore.unibas.ch/

k	1	10	100	1000	10000	100000
Coverage - Sum	**1726**	1674	1615	1468	1373	1527

Table 5.1: Comparison between different amounts of clusters - coverage of 2532 satisficing planning tasks

using $k = 100$ clusters, as shown in Table 5.2. It is interesting to note that $k = 100000$ is able to achieve a higher coverage than $k = \{1000, 10000\}$. This is due to tasks that generate less than 100000 states are basically still using random exploration comparable to $k = 1$ as not all of the clusters have been assigned a state yet, which means not having to calculate distances when inserting new nodes. What we did manage to achieve though, is to reduce the amount of expansions with $k = \{100, 1000\}$ for some of the hard planning tasks, that required more than 10^5 expansions when using the random open list with $k = 1$, as shown in Figure 5.1 and Figure 5.2. This trend indicates that clustering with the right amount of clusters actually may reduce the amount of expansions required for some difficult tasks, when compared to a random open list and therefore has the potential of being a an effective approach to exploring the state space if there weren't so many unsolved tasks. The issue with this trend is that it is possible that planning tasks which would have required more expansions when using Cluster-based open list couldn't be solved at all due to the implementation being inefficient and therefore reaching the search time limit of three minutes. We can see the significantly greater amount of unsolved tasks on the outer side of the graphs. The performance issues mainly come from having to calculate the distance to every cluster for every node that is being inserted into the open list. Finding the cluster with minimal distance to the node that is being inserted already uses about 20% of CPU time of the entire search algorithm when using $k = 10$. This increases to about 78% for $k = 1000$. Another but significantly less impactful factor is that the creation of the bit masks for every state also uses more CPU time than we anticipated. This effect can also be seen in the Figure 5.3, Figure 5.4 and Figure 5.5, where we compare the expansions per second ratio between different amounts of clusters.
In summary, we can see a trend of the Cluster-based open list being able to reduce the amount of expansions in hard to solve planning tasks when using $k = \{100, 1000\}$ compared to random exploration with $k = 1$. This trend could come from the fact that because of the inefficient implementation, planning tasks that would have required more expansions could not be solved at all by the Cluster-based open list. The performance issues come from having to calculate the distance to every cluster for every node that is being inserted into the open list.

5.2 Determining the Parameters for K-Means Reclustering

Clustering as described in Chapter 3 showed a trend of reducing the amount of expansions required to find a goal state. Indicating that if we clustered more effectively, reducing the entropy within clusters even further, we would need even fewer expansions. This is the expectation going into the following experiments where we used the Cluster-based open list with k-means reclustering as described in Chapter 4. As a consequence of clustering for a specific amount of time every interval, thus blocking the search algorithm for the entire duration, we expect the expansions per time to reduce even further, thus giving us an even lower coverage. Before we compare with and without reclustering to each other, we will compare different parameters for reclustering to find those that work best. For this purpose we first tried to determine a good time limit l after which we stop to recluster with a static time interval of one second and a static amount of clusters $k = 100$. The decision on the amount of clusters is somewhat arbitrary but comes as seemingly reasonable compromise between coverage and the usefulness of clustering. We chose the time interval $i = 1$ fully arbitrarily in order to make the testing of the following time limits $l = \{0.1, 0.2, 0.5, 1, 2\}$ (seconds) consistent. As expected the coverage has become even lower compared to not reclustering as we are using even more time to recluster. See Table 5.3. What is very interesting to note is that a the longest time limit for reclustering $l = 2$ resulted in the highest coverage indicating that the amount of expansions must have been reduced significantly for some of these additionally covered planning tasks, when

Coverage	No Reclustering $k = 100$	Random ($k = 1$)	With Reclustering $k = 100$	Single FF	type-based(ff, g())
airport (50)	**30**	**30**	26	**30**	**30**
assembly (30)	**30**	**30**	28	**30**	**30**
barman-sat11-strips(20)	0	**7**	0	4	2
barman-sat14-strips(20)	**0**	**0**	**0**	**0**	**0**
blocks(35)	**35**	**35**	32	**35**	**35**
cavediving-14-adl(20)	**7**	**7**	4	**7**	**7**
childsnack-sat14-strips(20)	**0**	**0**	**0**	**0**	**0**
citycar-sat14-adl(20)	**1**	0	0	0	0
depot(22)	13	**14**	9	**14**	**14**
driverlog(20)	16	18	14	17	**19**
elevators-sat08-strips(30)	**30**	**30**	24	**30**	**30**
elevators-sat11-strips(20)	10	13	4	**16**	11
floortile-sat11-strips(20)	4	**7**	1	4	6
floortile-sat14-strips(20)	**2**	**2**	1	**2**	**2**
freecell(80)	76	76	66	**77**	76
ged-sat14-strips(20)	10	13	0	**18**	14
grid(5)	4	**5**	4	4	4
gripper(20)	**20**	**20**	**20**	**20**	**20**
hiking-sat14-strips(20)	19	**20**	10	**20**	**20**
logistics00(28)	**28**	**28**	**28**	**28**	**28**
logistics98(35)	20	**24**	12	**24**	23
maintenance-sat14-adl(20)	4	5	0	10	**12**
miconic(150)	**150**	**150**	**150**	**150**	**150**
miconic-fulladl(150)	133	**136**	130	133	134
miconic-simpleadl(150)	**150**	**150**	**150**	**150**	**150**
movie(30)	**30**	**30**	**30**	**30**	**30**
mprime(35)	28	**29**	28	**29**	**29**
mystery(30)	**18**	17	16	17	**18**
nomystery-sat11-strips(20)	**10**	9	8	9	9
openstacks(30)	27	**30**	24	27	**30**
openstacks-sat08-adl(30)	**30**	**30**	24	**30**	**30**
openstacks-sat08-strips(30)	**30**	**30**	24	**30**	**30**
openstacks-sat11-strips(20)	10	**12**	4	10	10
openstacks-sat14-strips(20)	**0**	**0**	**0**	**0**	**0**
openstacks-strips(30)	26	**27**	23	**27**	**27**
optical-telegraphs(48)	2	**3**	2	**3**	**3**
parcprinter-08-strips(30)	**30**	**30**	**30**	**30**	**30**
parcprinter-sat11-strips(20)	**20**	**20**	**20**	**20**	**20**
parking-sat11-strips(20)	7	**10**	0	9	**10**
parking-sat14-strips(20)	0	0	0	**2**	0
pathways(30)	10	**14**	6	9	**14**
pathways-noneg(30)	10	**14**	6	9	**14**
pegsol-08-strips(30)	**30**	**30**	29	**30**	**30**
pegsol-sat11-strips(20)	**20**	**20**	19	**20**	**20**
philosophers(48)	24	35	12	**44**	35
pipesworld-notankage(50)	29	**38**	26	27	34
pipesworld-tankage(50)	22	**26**	14	21	23
psr-large(50)	16	**18**	11	14	16
psr-middle(50)	40	**42**	31	39	**42**
psr-small(50)	**50**	**50**	**50**	**50**	**50**
rovers(40)	19	22	17	23	**24**
satellite(36)	24	25	17	**26**	25
scanalyzer-08-strips(30)	24	26	21	25	**27**
scanalyzer-sat11-strips(20)	15	**16**	11	15	**16**
schedule(150)	37	**50**	21	30	48
sokoban-sat08-strips(30)	25	26	18	**27**	26
sokoban-sat11-strips(20)	15	16	9	**17**	**17**
storage(30)	19	**20**	18	18	18
tetris-sat14-strips(20)	**1**	**1**	0	0	**1**
thoughtful-sat14-strips(20)	**12**	11	**12**	8	10
tidybot-sat11-strips(20)	15	**16**	13	**16**	14
tpp(30)	15	15	10	**16**	15
transport-sat08-strips(30)	15	**16**	12	15	**16**
transport-sat11-strips(20)	**0**	**0**	**0**	**0**	**0**
transport-sat14-strips(20)	**0**	**0**	**0**	**0**	**0**
trucks(30)	14	12	10	14	**17**
trucks-strips(30)	15	15	12	14	**17**
visitall-sat11-strips(20)	2	4	0	3	**6**
visitall-sat14-strips(20)	**0**	**0**	**0**	**0**	**0**
woodworking-sat08-strips(30)	15	**23**	11	14	19
woodworking-sat11-strips(20)	2	**8**	1	2	4
zenotravel(20)	**20**	**20**	14	**20**	**20**
Sum 2532	1615	**1726**	1377	1662	1711

Table 5.2: Comparison of coverage on all domains

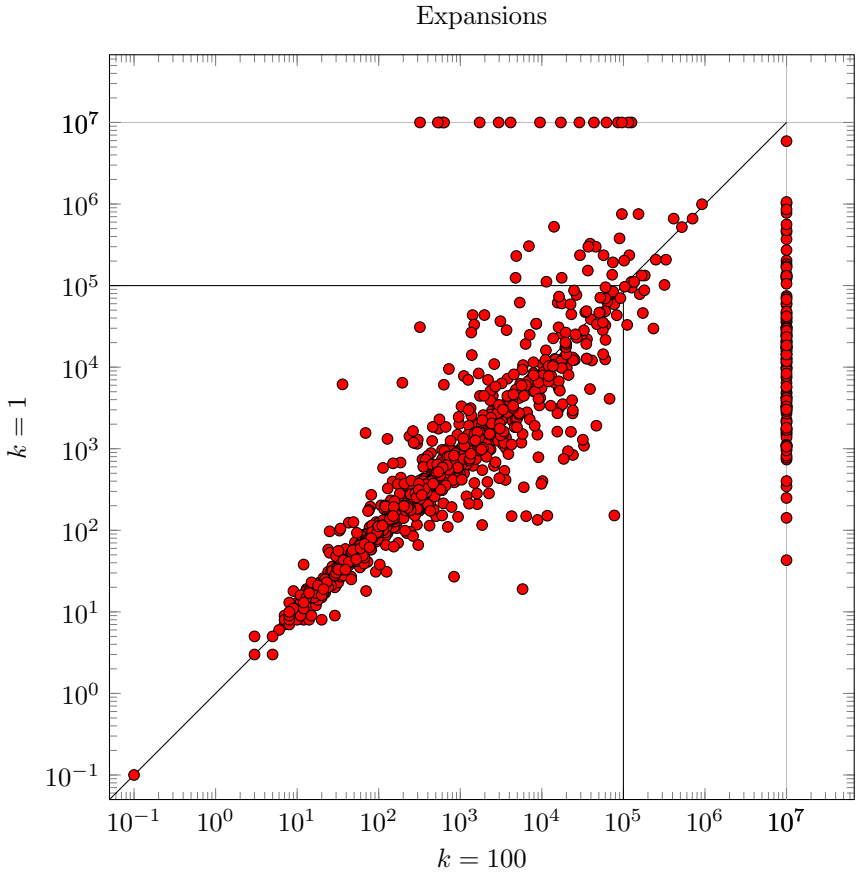

Figure 5.1: Comparison of the amount of expansions with $k = \{1, 100\}$

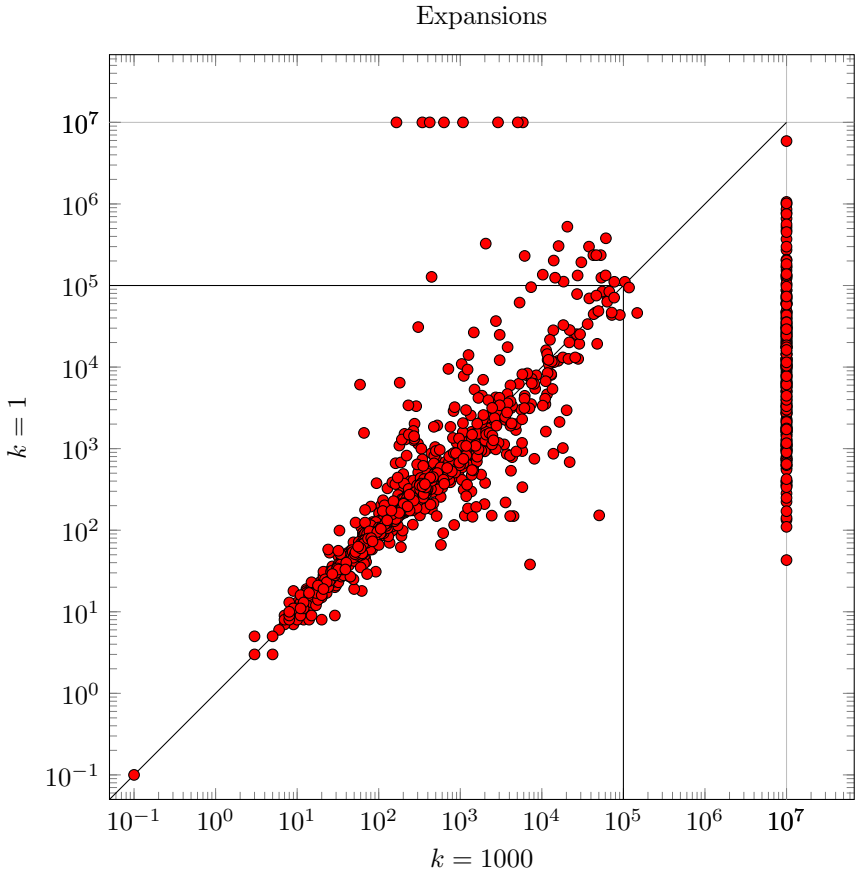

Figure 5.2: Comparison of the amount of expansions with $k = \{1, 1000\}$

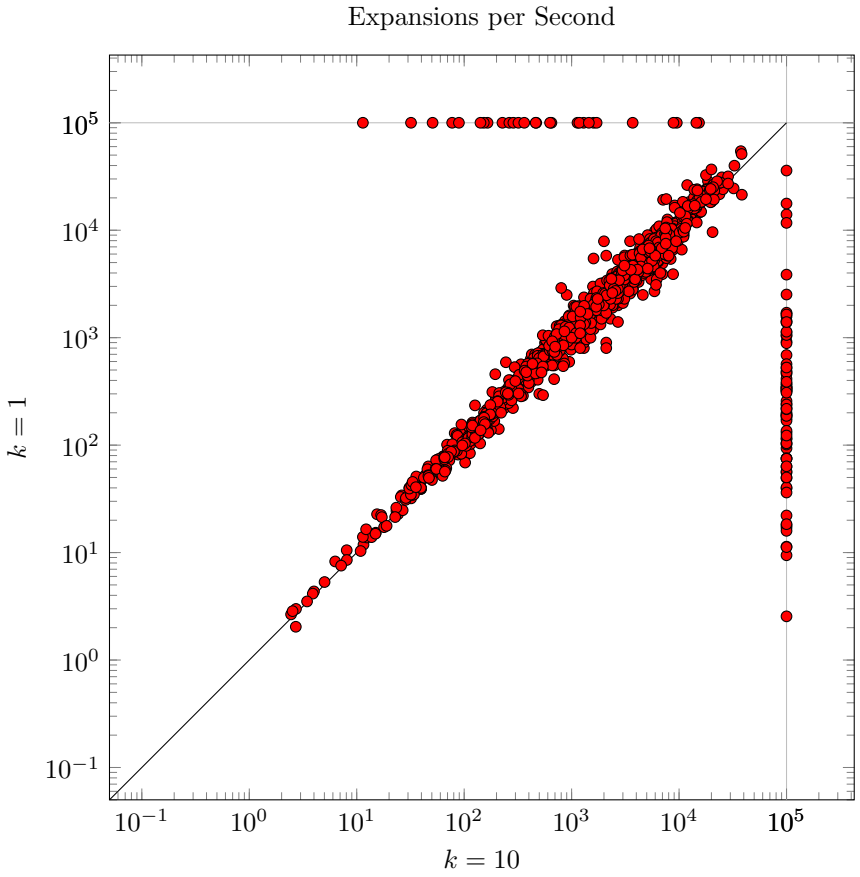

Figure 5.3: Comparison of the amount of expansions per second with $k = \{1, 10\}$

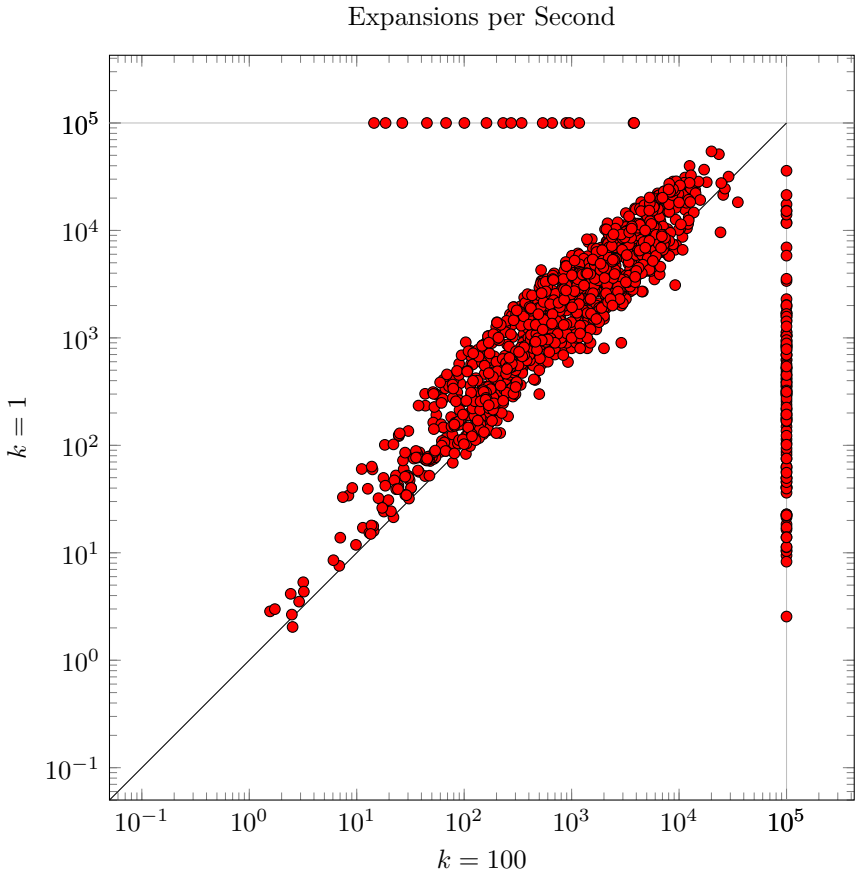

Figure 5.4: Comparison of the amount of expansions per second with $k = \{1, 100\}$

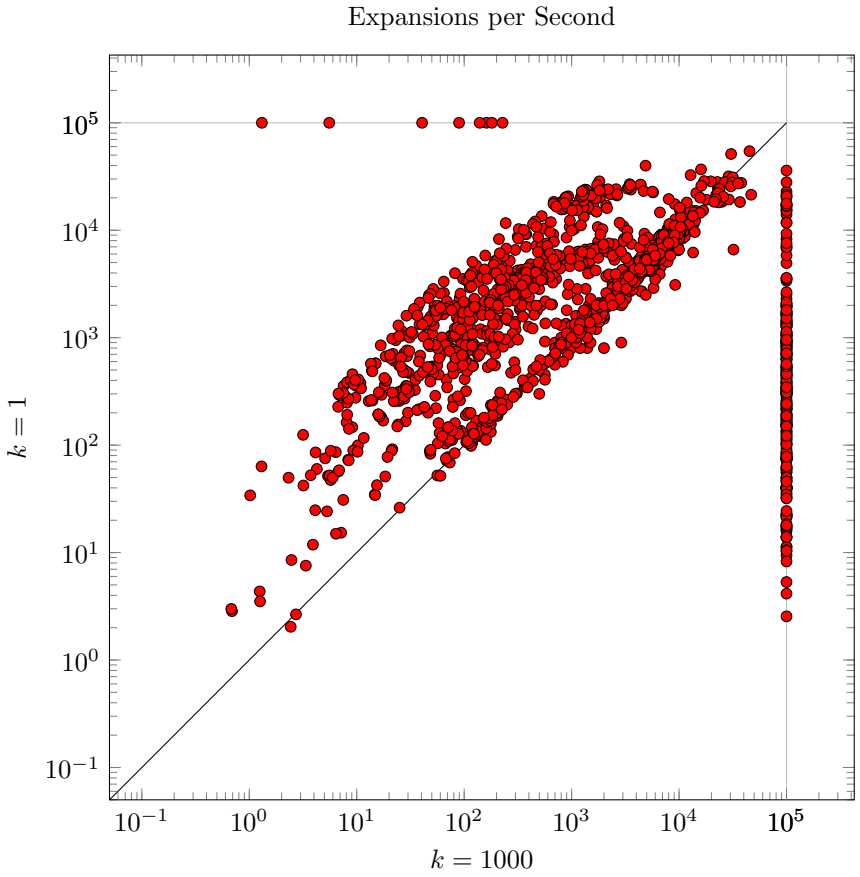

Figure 5.5: Comparison of the amount of expansions per second with $k = \{1, 1000\}$

reclustering for a longer time. This is confirmed in Figure 5.6 where planning tasks show a trend of benefiting from longer reclustering times, indicating that reclustering is effective at reducing entropy within clusters and therefore making exploration more diverse.

l	0.1	0.2	0.5	1	2
Coverage - Sum	1460	1454	1455	1447	**1461**

Table 5.3: Comparison between different time limits l with time interval $i = 1$ and $k = 100$ clusters - Coverage of 2532 satisficing planning tasks

Due to the highest coverage with $l = 2$ seconds as the time limit we chose $l = 2$ as the static parameter for our following experiments, as it also promises the highest effect on reducing entropy within clusters. The amount of clusters of $k = 100$ stayed the same and we went on to test different time intervals $i = \{0.1, 0.2, 0.5, 1, 2\}$. We expect a lower coverage for lower time intervals as we use more time to recluster. This holds true as we can see in Table 5.4. We do expect a higher amount of expansions for higher time intervals for commonly solved planning tasks, assuming the reclustering to be effective at reducing entropy within clusters.

i	0.1	0.2	0.5	1	2
Coverage - Sum	1340	1377	1432	1458	**1486**

Table 5.4: Comparison between different time intervals i with time limit $l = 2$ and $k = 100$ clusters - Coverage of 2532 satisficing planning tasks

As we can see in Figure 5.7 this is the case for commonly solved hard planning tasks, therefore indicating that reclustering could be a more effective way of exploration, if we didn't have the problem of the high run time.

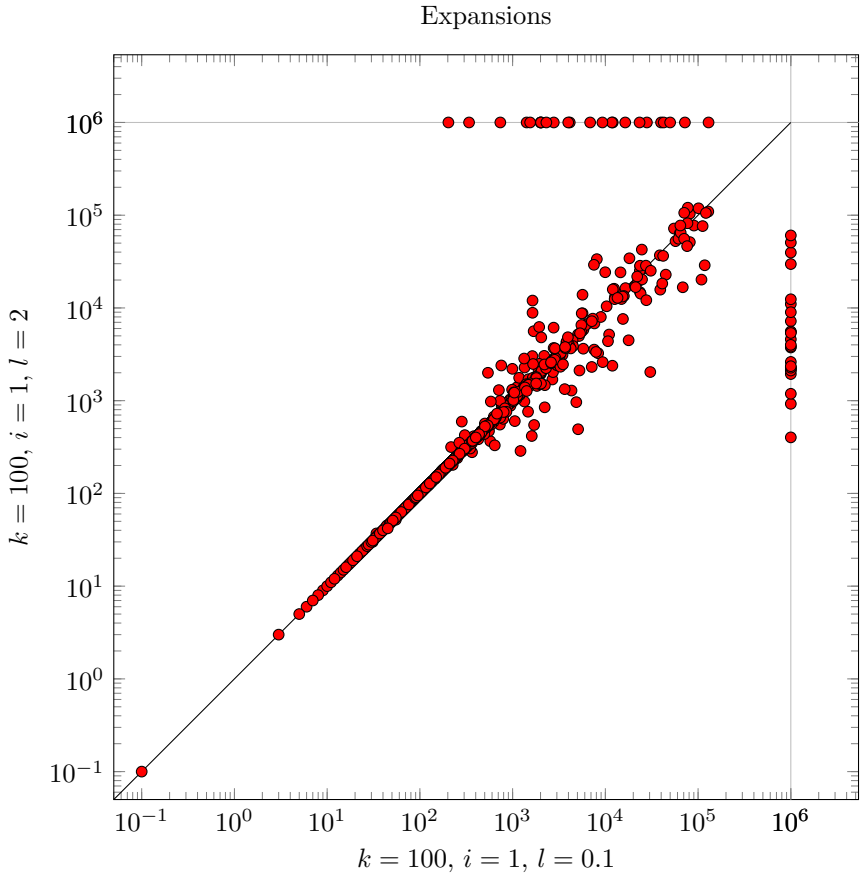

Figure 5.6: Comparison of the amount of expansions with time limits $l = \{0.1, 2\}$

Again the trend of reduced expansions for hard planning tasks could come from the fact that planning tasks that would have used more expansions could not be solved at all because of the search time limit being reached. For obvious reasons the expansions per second go down when we recluster more often as shown in Figure 5.8. To decide on which time interval to use for our comparison between using k-means for reclustering and not reclustering at all, we look at Figure 5.9 which shows some tendency to reduce the amount of expansions when using $i = 0.1$ compared to $i = 0.5$ for some of the hard planning tasks. Using $i = 0.1$ over $i = 0.2$, however shows no such tendency, as illustrated in Figure 5.10, but effectively doubles our run time. This is why we decided on using $i = 0.2$ and $l = 2$ as our default parameters for reclustering, to get the most effect out of reclustering while not punishing our run time too much. If not stated otherwise when we refer to Cluster-based open list *with* reclustering we will assume the use of these parameters.

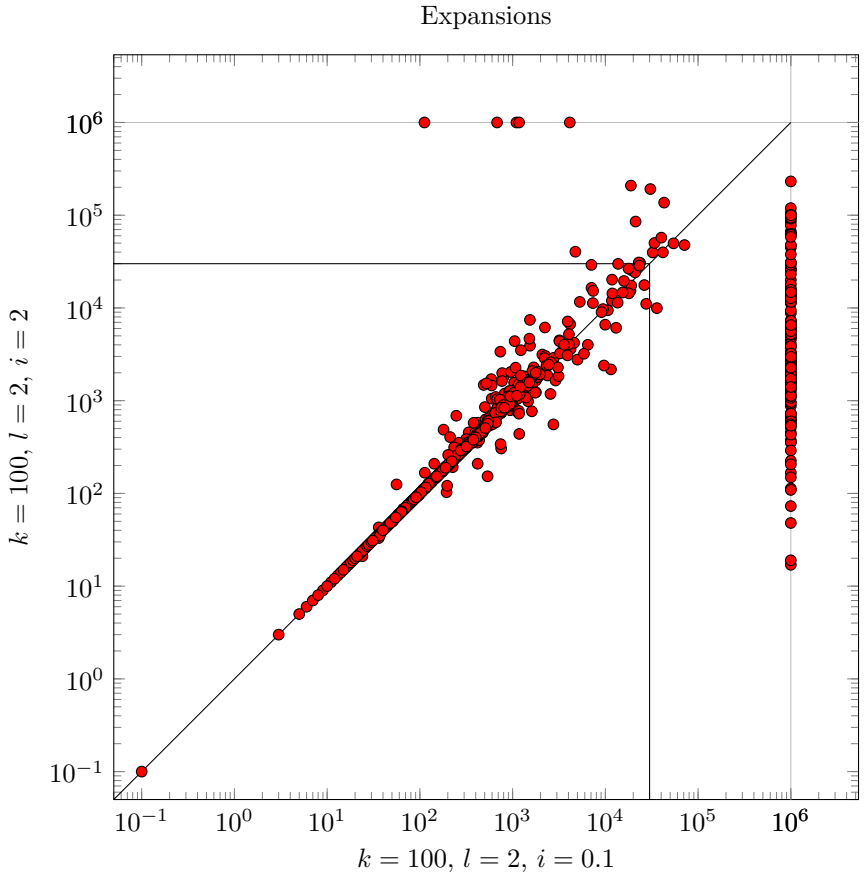

Figure 5.7: Comparison of the amount of expansions with time intervals $i = \{0.1, 2\}$

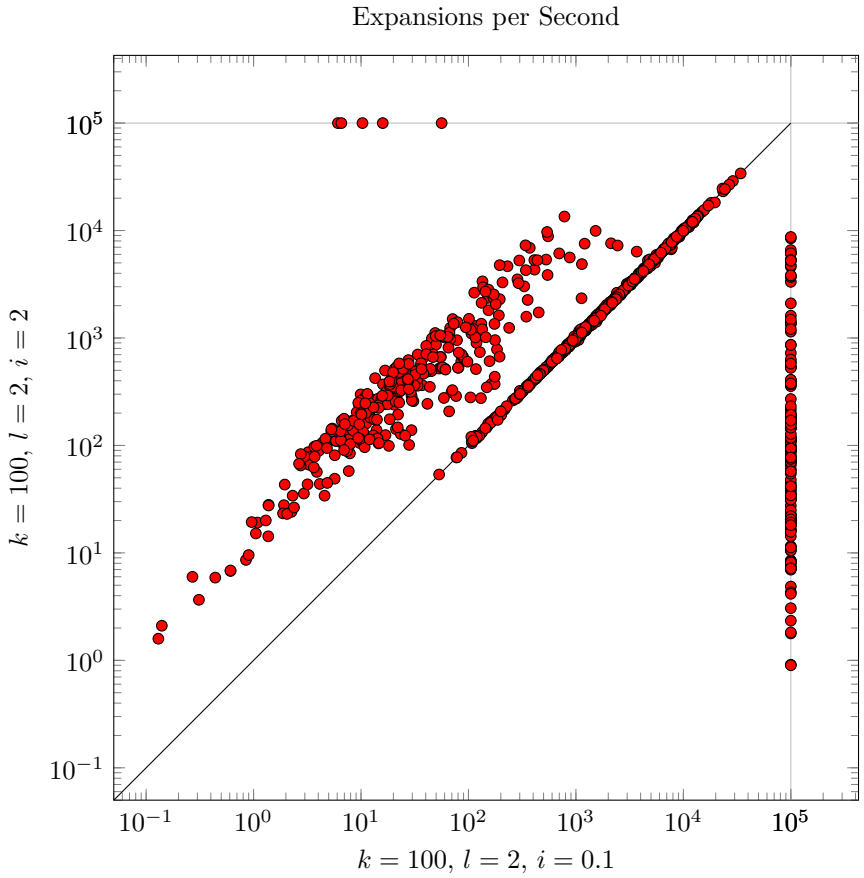

Figure 5.8: Comparison of the amount of expansions per second with time intervals $i = \{0.1, 2\}$

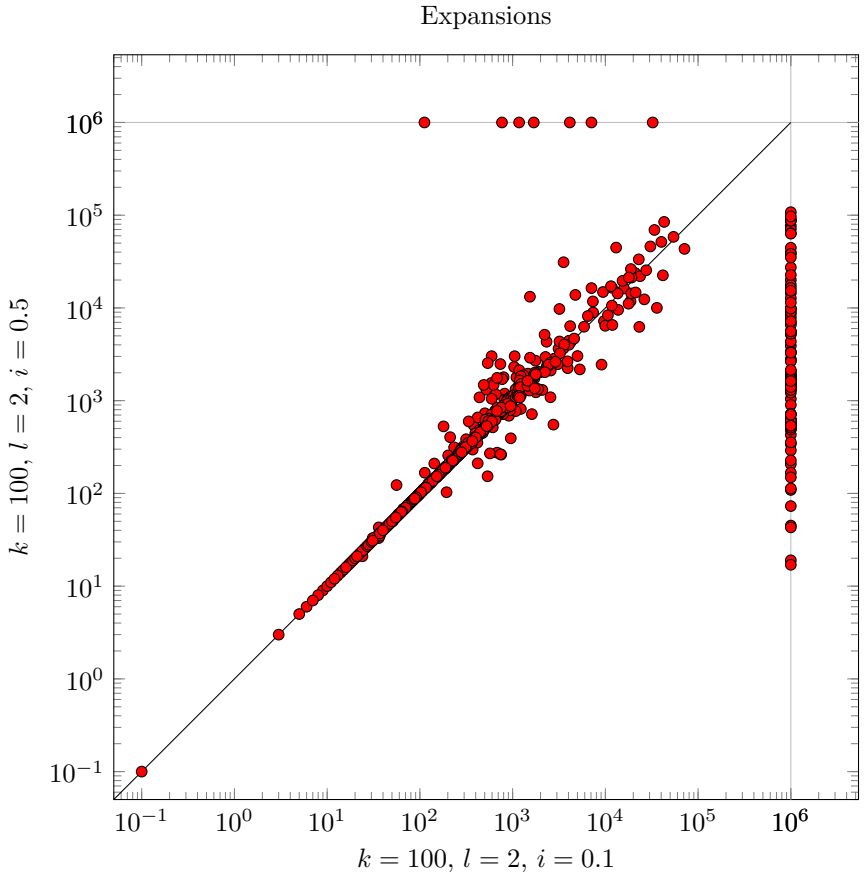

Figure 5.9: Comparison of the amount of expansions with time intervals $i = \{0.1, 0.5\}$

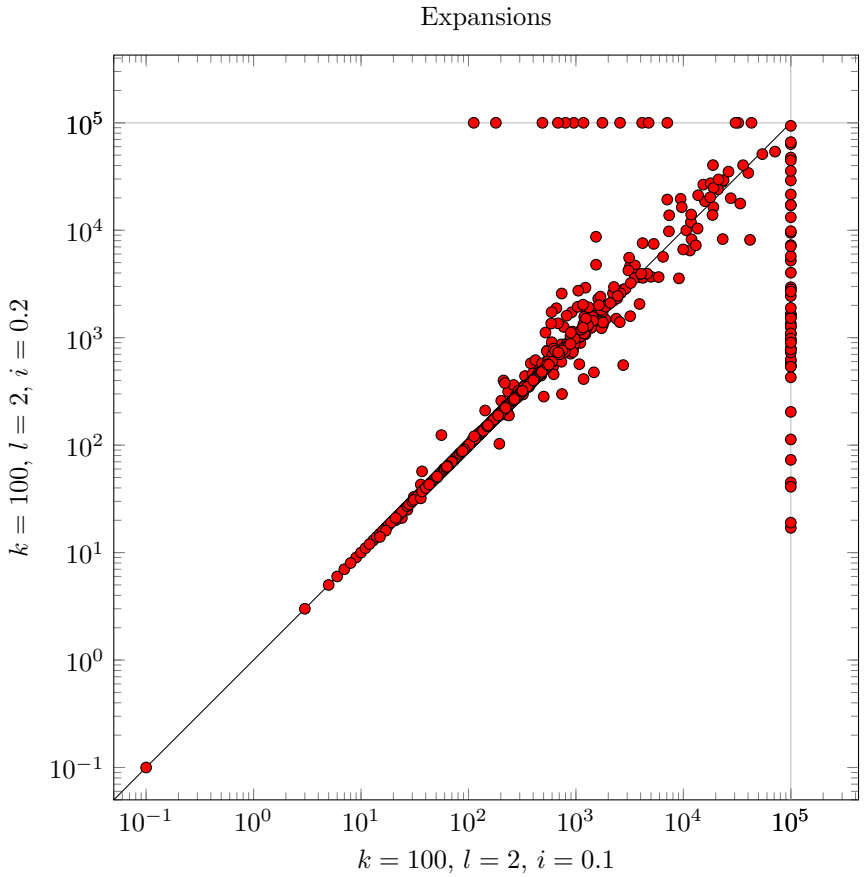

Figure 5.10: Comparison of the amount of expansions with time intervals $i = \{0.1, 0.2\}$

5.3 Comparing Reclustering with not Reclustering

As expected we get a much lower coverage when we recluster with the above mentioned parameters of time limit $l = 2$ and time interval $i = 0.2$, due to the amount of time spent on doing so. This is confirmed in Table 5.5. There is one domain *thoughtful-sat14-strips* where reclustering is actually on part with not reclustering and where both of the implementations reached a higher coverage than random exploration. See Table 5.2.

	No Reclustering	With Reclustering
Coverage - Sum	**1615**	1372

Table 5.5: Comparison between reclustering and not reclustering with 100 clusters - coverage of 2532 satisficing planning tasks

As more frequent reclustering has shown a trend of requiring fewer expansions for commonly solved tasks we expect the same when comparing reclustering ($l = 2$, $i = 0.2$) to not reclustering at all. This is confirmed in Figure 5.11, which even shows this trend even for easier planning tasks.

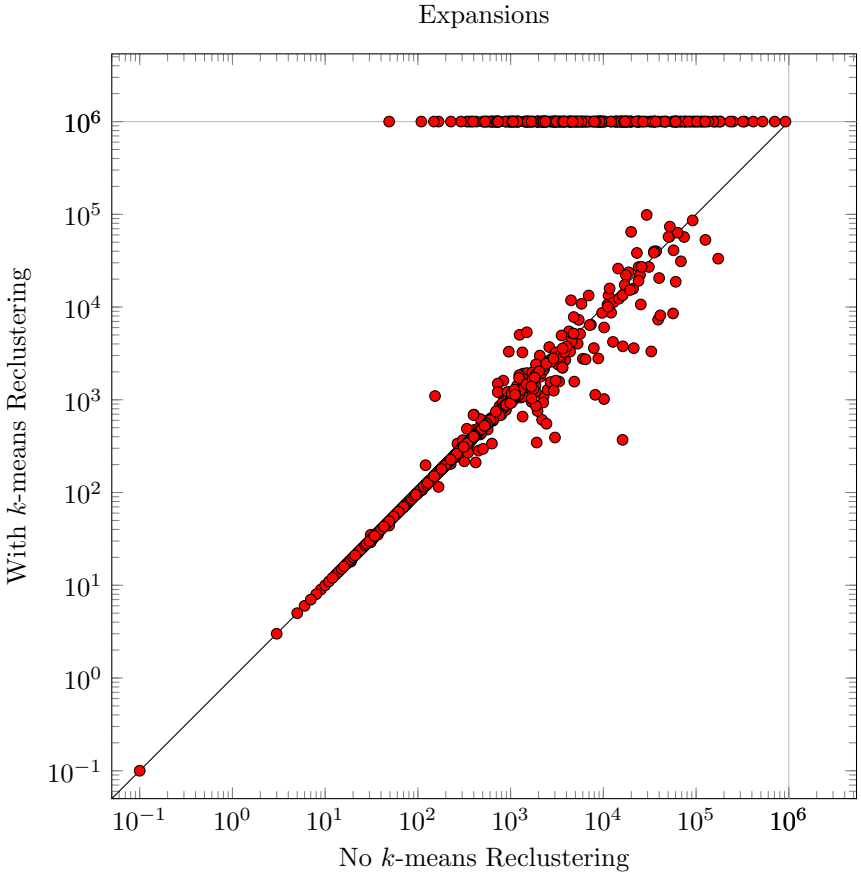

Figure 5.11: Comparison of the amount of expansions when reclustering vs not reclustering

5.4 Comparing to GBFS without Exploration

In this section we compare GBFS with FF using both configurations of the Cluster-based open list (with and without reclustering) as the exploration open list and using no exploration at all (Single FF). Again the coverage of GBFS + Cluster-based open list can not match Single FF as we can see in Table 5.6, due to the inefficiencies we explained earlier. There is the same interesting domain *thoughtful-sat14-strips* which got the lowest coverage by Single FF, but was solved most effectively when using Cluster-based open list (with and without reclustering) for exploration. See Table 5.2.

	No Reclustering	With Reclustering	Single FF
Coverage - Sum	1615	1377	**1662**

Table 5.6: Comparison between Cluster-based with reclustering and GBFS using FF without exploration - Coverage of 2532 satisficing planning tasks

There is a clear trend of Single FF requiring more expansions for harder planning tasks when compared to both Cluster-based configurations as illustrated in Figure 5.12 and Figure 5.13. We have to mention again that this trend could come from GBFS + Cluster-based exploration not being able to solve the hard planning tasks that would have required significantly more expansions than Single FF because the implementation is inefficient and therefore reaches the search time limit.

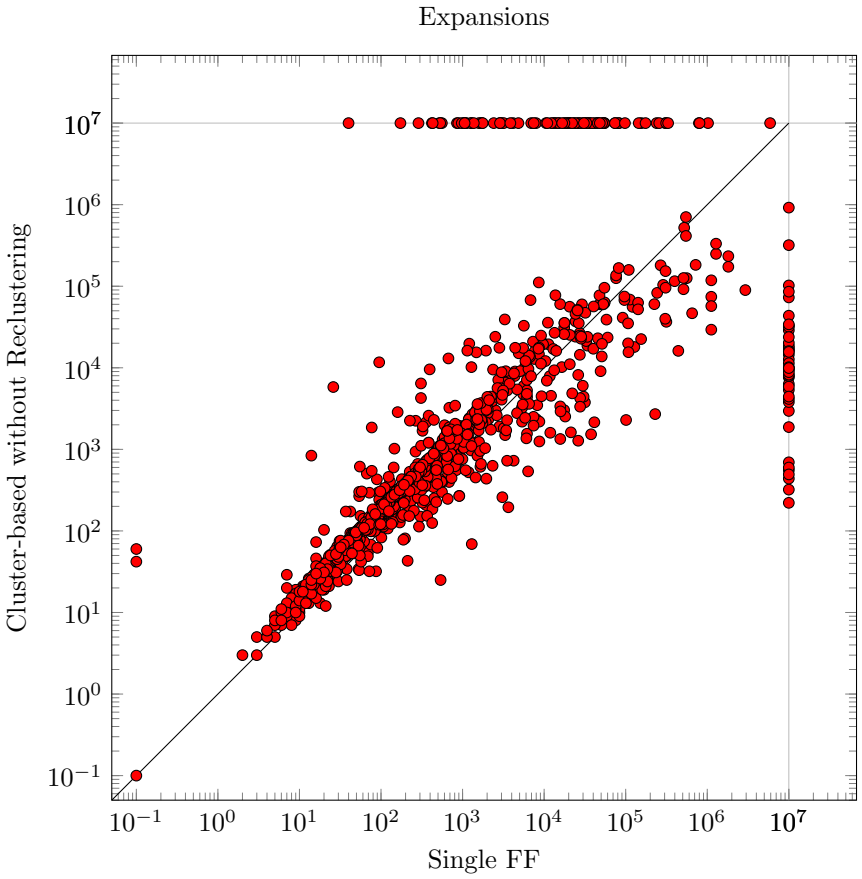

Figure 5.12: Comparison of expansions between Single FF and Cluster-based exploration without Reclustering

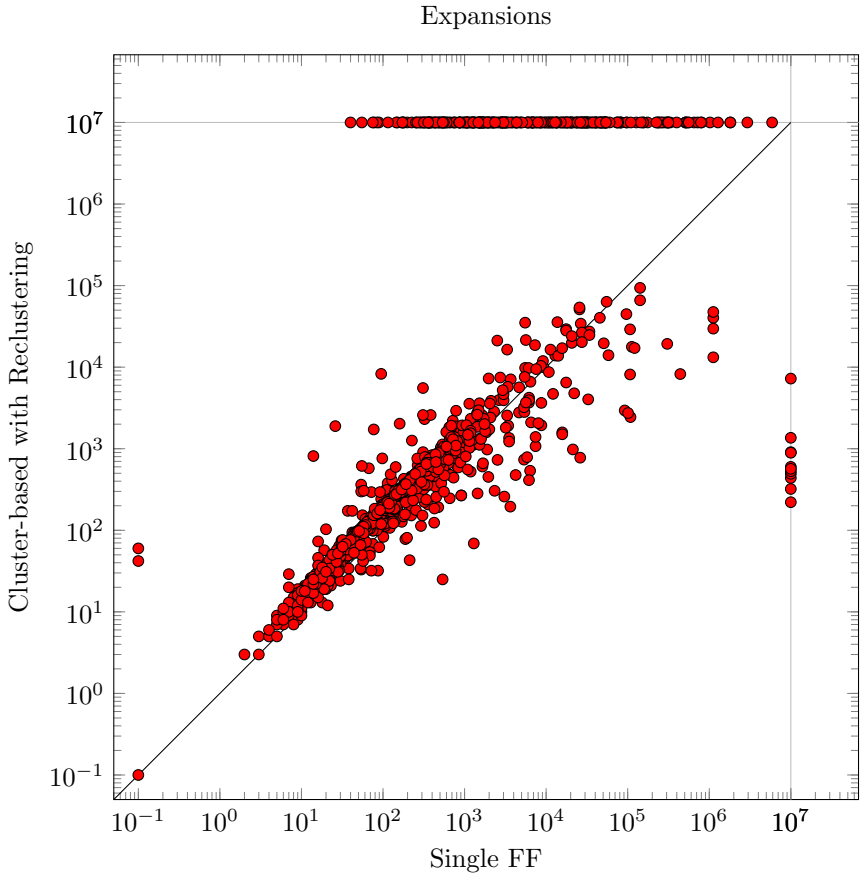

Figure 5.13: Comparison of expansions between Single FF and Cluster-based exploration with Reclustering

5.5 Comparing to Type-based Exploration

When comparing our implementation with the Type-based open list by Xie et al. [2014] it is not at all surprising that we were not able to compete with their coverage due to our inefficient insertion process we explained earlier. We mentioned the domain *thoughtful-sat14-strips* before. Again it is worth mentioning because both configurations of Cluster-based open list (with and without reclustering) were able to beat the random configuration $k = 1$, but the Type-based open list was not.

Interestingly there is the same trend of reduced amount of expansions compared to Type-based with both configurations, with and without reclustering, for some commonly solved, hard planning tasks, as illustrated in Figure 5.14 and Figure 5.15. The trend has the same potential flaw that planning tasks that would have required more expansions with Cluster-based exploration, may not have been solved at all. What came somewhat as a surprise is that Type-based exploration reached a lower coverage than random exploration using the Cluster-based open list with $k = 1$ as shown in Table 5.2.

	No Reclustering	With Reclustering	Type-based
Coverage - Sum	1615	1377	**1711**

Table 5.7: Comparison between Cluster-based without reclustering, with reclustering and Type-based - Coverage of 2532 satisficing planning tasks

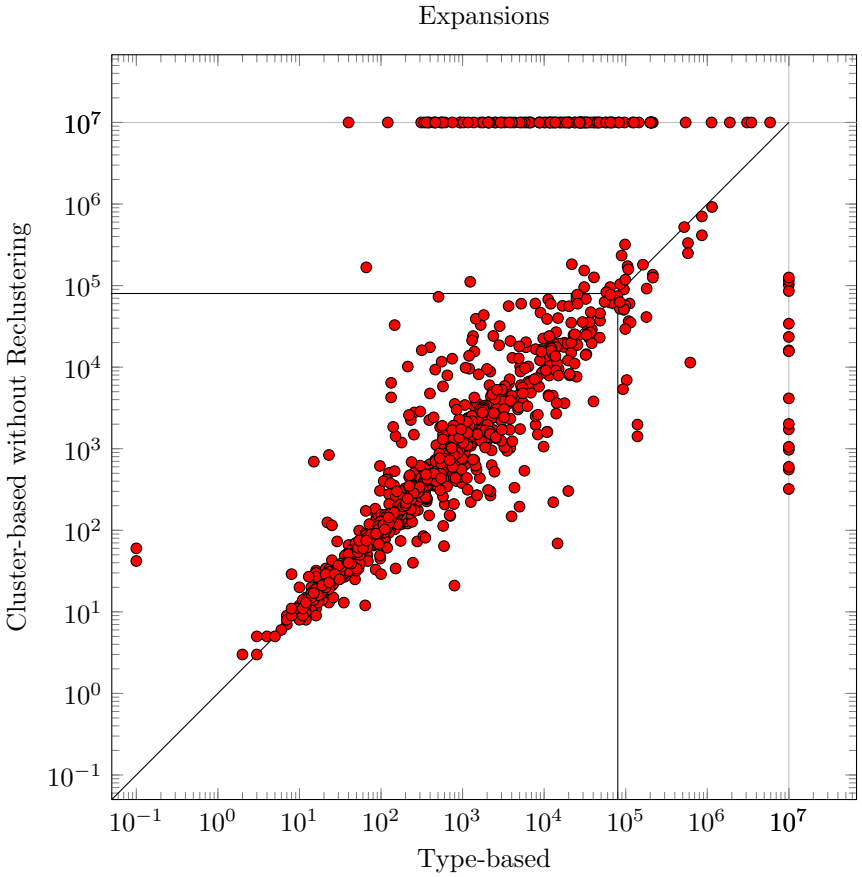

Figure 5.14: Comparison of expansions between Type-based and Cluster-based exploration with Reclustering

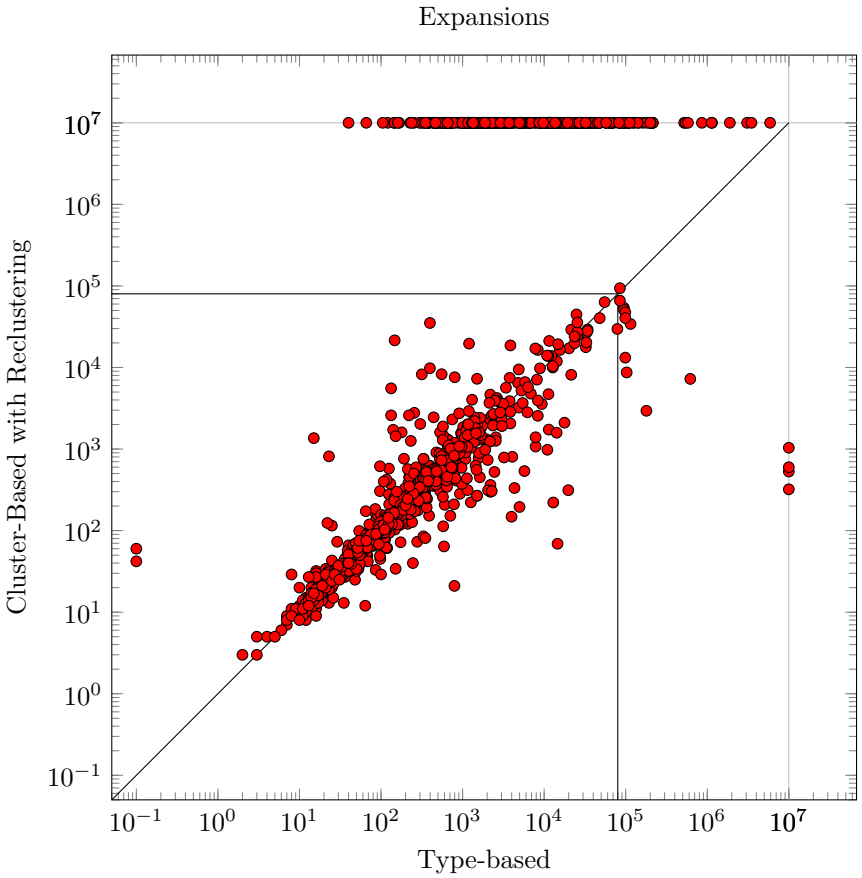

Figure 5.15: Comparison of expansions between Type-based and Cluster-based exploration with Reclustering

6 Conclusion

We discussed how the Cluster-based open list can be used in conjunction with GBFS in order to introduce state space exploration to negate mistakes made by misleading heuristics. Furthermore we elaborated on how clustering states based on the hamming distance of their binary representation may be able to reduce entropy within clusters allowing for a more diverse and therefore more effective exploration when expanding states from a randomly chosen cluster.
The conducted experiments showed a trend of reducing the amount of expansions for hard satisficing planning tasks when compared to random exploration. When compared to Single FF there was a clear trend of fewer required expansions when we add exploration using the Cluster-based open list. Cluster-based exploration was also able to reduce the amount of expansions for some hard planning tasks when compared to Type-based exploration. The problem with these trends is that the inefficient implementation might not have been able to solve the hard planning tasks that would have required more expansions to solve. We were able to see a more promising trend of reclustering using k-means being able to reduce the amount of expansions even for easier planning tasks compared to the configuration without reclustering.
We weren't able to reach a higher coverage of the tested planning tasks compared to random exploration, Single FF or the Type-based open list, because of the inefficiency of the implemented algorithm, caused mainly by the numerous distance calculations between new states and all clusters. Because of the k-means algorithm adding an additional computational overhead, the Cluster-based open list became even less effective.
As the trend of reduced expansions for hard to solve planning tasks could have occurred due to GBFS + Cluster-based being unable to solve those tasks that would have required more expansions, further testing with a more efficient implementation or a higher search time limit is required to confirm this trend to be consistent.

6.1 Future Work

In order to increase the speed of the Cluster-based open list, one could try only doing clustering runs for a large amount of nodes at once instead of assigning every new node to a cluster right at the time of insertion.
We could also try and optimize the algorithm by introducing parallelism to the process of finding the cluster with minimal distance to the node that is to be inserted, by forking k child processes that calculate the node's distance to the k clusters. The parent process then decides, based on the results that the children processes have returned, in which cluster the new node should be inserted to.
If we are able to make Cluster-based open list significantly faster, and the trend of reducing the amount of expansions for harder planning tasks turns out to be consistent, it has the potential to be an effective approach to diversifying GBFS with efficient state space exploration and consequently produce a high coverage of satisficing planning tasks.

A
Appendix

```
template<class Entry>
class ClusterBasedOpenList : public OpenList<Entry> {
       struct StateTuple {
        Entry entry;
        vector<bool> bit_mask;
        StateTuple(Entry e, vector<bool> b) : entry(e), bit_mask(b) {}
    };

    class Bucket {
        vector<StateTuple> states;
        vector<int> facts_count;
        vector<double> facts_mean;

        public:

        Bucket () {
            facts_count = vector<int>(utils::get_num_facts());
            facts_mean = vector<double>(utils::get_num_facts());
        }

        bool operator <(const Bucket &other) const {
            return states.size() < other.states.size();
        }

        double get_distance(const vector<bool> &bit_mask) const {
            assert(static_cast<int>(bit_mask.size()) == utils::get_num_facts());
            double sum = 0;
            for (size_t i = 0; i < bit_mask.size(); i++) {
                double diff = abs(facts_mean[i] - bit_mask[i]);
                sum += diff;
            }
            return sum;
        }

        int size() {
            return states.size();
        }

        void clear() {
            states.clear();
            fill(facts_count.begin(), facts_count.end(), 0);
            fill(facts_mean.begin(), facts_mean.end(), 0);
        }

        void clear_but_keep_means() {
```

```
            states.clear();
            fill(facts_count.begin(), facts_count.end(), 0);
        }

        void insert(StateTuple &state, bool keepMeans) {
            states.push_back(state);
            for (int i = 0; i < utils::get_num_facts(); ++i) {
                if (state.bit_mask[i]) {
                    facts_count[i]++;
                }
                if (!keepMeans) {
                    facts_mean[i] = static_cast<double>(facts_count[i]) / size();
                }
            }
        }

        Entry pop(int pos) {
            assert(utils::in_bounds(pos, states));
            StateTuple state = utils::swap_and_pop_from_vector(states, pos);
            Entry &result = state.entry;
            vector<bool> &bit_mask = states[pos].bit_mask;
            if (!empty()) {
                for (int i = 0; i < utils::get_num_facts(); i++) {
                    if (bit_mask[i]) {
                        facts_count[i]--;
                    }
                    facts_mean[i] = static_cast<double>(facts_count[i]) / states.size();
                }
            } else {
                clear();
            }
            return result;
        }

        StateTuple pop_end_state_tuple() {
            StateTuple result = states.back();
            states.pop_back();
            return result;
        }

        bool empty() {
            return states.empty();
        }

        vector<double> &get_facts_mean() {
            return facts_mean;
        }

        vector<int> &get_facts_count() {
            return facts_count;
        }

        void recalculate_mean() {
            if (!states.empty()) {
                for (int i = 0; i < utils::get_num_facts(); i++) {
                    facts_mean[i] = static_cast<double>(facts_count[i]) / states.size();
                }
            } else {
                generate(facts_mean.begin(), facts_mean.end(), []() {
                    return static_cast<double>(rand()) / (RAND_MAX);
                });
```

```
            }
        }
    };

    int get_min_index(const vector<bool> &bit_mask) {
        double min_distance = numeric_limits<double>::max();
        vector<int> min_indexes;
        for (int i = 0; i < static_cast<int>(buckets.size()); i++) {
            double distance = buckets[i].get_distance(bit_mask);
            if (distance < min_distance) {
                min_distance = distance;
                min_indexes.clear();
                min_indexes.push_back(i);
            }
            if (distance == min_distance) {
                min_indexes.push_back(i);
            }
        }
        if (min_indexes.size() == 1) {
            assert(utils::in_bounds(min_indexes[0], buckets));
            return min_indexes[0];
        } else {
            assert((int)min_indexes.size() > 0);
            int index = (*g_rng())((int)min_indexes.size());
            assert(utils::in_bounds(min_indexes[index], buckets));
            return min_indexes[index];
        }
    }

    void k_means() {
        utils::CountdownTimer stop_timer(stop_time);
        while (!stop_timer.is_expired()) {
            for (Bucket &bucket : buckets) {
                bucket.clear_but_keep_means();
            }
            for (StateTuple &state : all_states) {
                int min_index = get_min_index(state.bit_mask);
                buckets[min_index].insert(state, true);
            }
            for (Bucket bucket : buckets) {
                bucket.recalculate_mean();
            }
        }
        sort(buckets.rbegin(), buckets.rend());
        while (buckets[num_filled - 1].empty()) {
            --num_filled;
        }
    }

    int num_buckets;
    double interval;
    double stop_time;
    vector<Bucket> buckets = vector<Bucket>(num_buckets);
    int num_filled = 0;
    utils::Timer timer;
    vector<StateTuple> all_states;

template<class Entry>
void ClusterBasedOpenList<Entry>::do_insertion(EvaluationContext &eval_context, const Entry &entry) {
    const GlobalState &state = eval_context.get_state();
    int index_to_insert = -1;
```

```
    vector<bool> bit_mask = utils::create_bit_mask(state);
    StateTuple stateTuple = StateTuple(entry, bit_mask);
    if (num_filled < num_buckets) {
        assert(utils::in_bounds(num_filled, buckets));
        index_to_insert = num_filled;
        num_filled++;
        if (num_filled == num_buckets && interval > 0) {
            timer.reset();
        }
    } else {
        index_to_insert = get_min_index(bit_mask);
    }
    buckets[index_to_insert].insert(stateTuple, false);
    all_states.push_back(stateTuple);
    if (interval > 0 && num_filled == num_buckets) {
        if (timer() > interval) {
            k_means();
            timer.reset();
        }
    }
}

template<class Entry>
Entry ClusterBasedOpenList<Entry>::remove_min(vector<int> *) {
    assert(num_filled > 0);
    int index = (*g_rng())(num_filled);
    assert(utils::in_bounds(index, buckets));
    Bucket &bucket = buckets[index];
    assert(bucket.size() > 0);
    int pos = (*g_rng())(static_cast<int>(bucket.size()));
    Entry result = bucket.pop(pos);
    if (bucket.empty()) {
        assert(utils::in_bounds(num_filled - 1, buckets));
        buckets[index] = buckets[num_filled - 1];
        buckets[num_filled - 1].clear();
        num_filled--;
    }
    return result;
}

template<class Entry>
bool ClusterBasedOpenList<Entry>::empty() const {
    return num_filled == 0;
}
```

These are only the most important methods of the C++ implementation. For the entire working code integrated into *Fast Downward* go to https://bitbucket.org/danielkillenberger/openlist-clustering.

B
Appendix

The following are the search options used for different parameters using the *Fast Downward* planning system:

Cluster-based without Reclustering and $k = 100$:

```
--search "eager(alt([single(ff(cost_type = one)), cluster_based(100)]), cost_type = one)"
```

Cluster-based with Reclustering, $i = 0.2$, $l = 2$ and $k = 100$:

```
--search "eager(alt([single(ff(cost_type = one)), cluster_based(100, 0.2, 2)]), cost_type = one)"
```

Cluster-based with only one cluster $k = 1$ (random exploration):

```
--search "eager(alt([single(ff(cost_type = one)), cluster_based(1)]), cost_type = one)"
```

GBFS with Single FF:

```
--search "eager(single(ff(cost_type = one)), cost_type = one)"
```

Type-based configuration:

```
--heuristic "hff=ff(cost_type=one)"
--search "eager(alt([single(hff), type_based(hff, g())]), cost_type = one)"
```

Bibliography

Bäckström, C. and Nebel, B. (1995). Complexity Results for SAS+ Planning. *Computational Intelligence*, 11(4).

Ghallab, M., Nau, D., and Traverso, P. (2004). *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers, 500 Sansome Street, San Francisco.

Hamming, R. W. (1950). Error Detecting and Error Correcting Codes. *The Bell System Technical Journal*, 29(2).

Helmert, M. (2006). The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26.

Hoffman, J. and Nebel, B. (2001). The FF Planning System: Fast Plan Generation Trough Heuristic Search. *Journal of Artificial Intelligence Research*, 14.

Pearl, J. (1984). *Heuristics - Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Publishin Company, California.

Richter, S. and Westphal, M. (2010). The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research*, 39.

Russel, S. J., Norvig, P., Canny, J. F., Malik, J. M., and Edwards, D. D. (1995). *Aritificial Intelligence A Modern Approach*. Prentice Hall, Englewood Cliffs, New Jersey 07632.

Valenzano, R., Sturtevant, N. R., Schaeffer, J., and Xie, F. (2014). A Comparison of Knowledge-Based GBFS Enhancements and Knowledge-Free Exploration. In *Twenty-Fourth International Conference on Automated Planning and Scheduling*.

Weld, D. S. (1999). Recent Advances in AI Planning. *AI Magazine*, 20(2).

Xie, F., Müller, M., Holte, R., and Imai, T. (2014). Type-based Exploration with Multiple Search Queues. In *AAAI'14 Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*.

UNIVERSITÄT BASEL

PHILOSOPHISCH-NATURWISSENSCHAFTLICHE FAKULTÄT

Declaration on Scientific Integrity
(including a Declaration on Plagiarism and Fraud)

Bachelor's / [redacted] Thesis *(Please cross out what does not apply)*

Title of Thesis *(Please print in capital letters)*:

Diversifying Greedy Best-First Search by Clustering States

First Name, Surname *(Please print in capital letters)*: Daniel Killenberger

Matriculation No.: 2013-059-647

I hereby 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.

In addition to this declaration, I am submitting a separate agreement regarding the publication of or public access to this work.

■ Yes ☐ No

Place, Date: Basel, 26.02.2017

Signature: D Killenberger

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

UNI
BASEL