# Contribution Title*

First Author[1[0000−1111−2222−3333]] and Fahiem Bacchus[2,3[1111−2222−3333−4444]]

[1] Princeton University, Princeton NJ 08544, USA
[2] Springer Heidelberg, Tiergartenstr. 17, 69121 Heidelberg, Germany
lncs@springer.com
http://www.springer.com/gp/computer-science/lncs
[3] ABC Institute, Rupert-Karls-University Heidelberg, Heidelberg, Germany
{abc,lncs}@uni-heidelberg.de

## 1 Introduction

## 2 Preliminaries

## 3 Find i-uip Clause

Classical CDCL SAT solvers analyze each conflict with 1-UIP resolution learning scheme to derive an asserting clause $C_1$. Even though prior studies shows that further resolutions on $C_1$ against the assertion trail $\phi$ cannot reduce the clause's literal block distance (LBD), the resolutions could potentially reduce the clause's size. Clause size is an important quality measurement because a smaller clause 1) consumes less memory, 2) requires less steps to force a literal and 3) decreases the size of future conflict clauses.

Inspired by both LBD and clause size, i-UIP resolution learning attempts to resolve away literals in $C_1$ against the assertion trail $\phi$ to minimize the clause size without increasing the clause's LBD. The goal of i-UIP learning is to find a clause $C_i$ whose literals are either the unique implication points in their respective decision levels or are directly implied by literals from a foreign decision level. Alg. 1 is a pseudo-code implementation of i-UIP learning.

The algorithm **i-UIP** computes $C_i$ from (line 2 to 20), and then returns the smaller clause between $C_1$ and $C_i$ (line 21 to 6). The algorithm first initializes $C_i$ with the minimized[] $C_1$. Next, the algorithm iterate through the decision levels of $C_i$ in descending order (line 4) and tries to find the unique implication point in each level (line 6). Since **i-UIP** needs to preserve $C_i$'s LBD during resolutions, before resolving away a target literal $p$, the algorithm preemptively checks whether the reasoning clause of $\neg p$ contains any literal $q$ from an unseen decision level (line 9). If such a literal $q$ exists, then the algorithm cannot find the unique implication point (UIP) at the current level $i$ because resolving away $p$ will introduce $q$ into the clause and increases LBD. One solution is to abandon level $i$ by reverting back to the state before any literal at level $i$ is resolved away (line 10), and then move on to the next level (line 11). We denote **i-UIP** learning with this solution as **Pure-i-UIP**. The final $C_i$ produced by **Pure-i-UIP**

* Supported by organization x.

**Algorithm 1 i-UIP**

**Require:** $C_1$ is a valid and minimized 1-UIP clause
**Require:** $\phi$ is a valid assertion trail
1: **procedure** **i-UIP**$(C_1, \phi)$
2: $\quad C_i \leftarrow C_1$ $\triangleright$ initialize i-UIP clause
3: $\quad$ DecisionLvs $\leftarrow$ Decision levels in $C_1$ in descending order
4: $\quad$ **for** $i \in$ DecisionLvs **do**
5: $\quad\quad L_i \leftarrow \{\, l \mid \text{level}(l) = i \wedge l \in C_i \,\}$
6: $\quad\quad$ **while** $|\text{unmarked}(L_i)| > 1$ **do**
7: $\quad\quad\quad p \leftarrow$ lit with the highest trail position in $L_i$
8: $\quad\quad\quad$ **if** $\exists q \in \text{Reason}(\neg p, \phi) \cdot \text{level}(q) \notin$ DecisionLvs **then**
9: $\quad\quad\quad\quad$ **if Pure-i-UIP then**
10: $\quad\quad\quad\quad\quad$ Unresolve literals at level $i$
11: $\quad\quad\quad\quad\quad$ Go to the next decision level $i$
12: $\quad\quad\quad\quad$ **else if Min-i-UIP then**
13: $\quad\quad\quad\quad\quad$ Mark$(p)$
14: $\quad\quad\quad\quad$ **end if**
15: $\quad\quad\quad$ **else**
16: $\quad\quad\quad\quad C_i \leftarrow (C_1) \bowtie \text{Reason}(\neg p, \phi)$
17: $\quad\quad\quad\quad$ Update$(L_i)$
18: $\quad\quad\quad$ **end if**
19: $\quad\quad$ **end while**
20: $\quad$ **end for**
21: $\quad$ **if** $|C_i| < |C_1|$ **then**
22: $\quad\quad$ **return** $C_i$
23: $\quad$ **else**
24: $\quad\quad$ **return** $C_1$
25: $\quad$ **end if**
26: **end procedure**

will contains exactly one literal for levels where the LBD-persevering UIP exist. However, **Pure-i-UIP** does not minimize the number of literals in the decision levels where the LBD-preserving UIP does not exist.

When **i-UIP** cannot resolve away a literal $p$ without increasing $C_i$'s LBD, instead of skipping the entire decision level $i$, a more practical solution **Min-i-UIP** will mark and keep $p$ in $C_i$ (line 13), and continue resolutions at level $i$. **Min-i-UIP** will ignore all the unsolvable literals and find the "local" unique implication point at every decision level. The algorithm terminates when there is exactly one unmarked literal left for each decision level of $C_i$, representing the local unique implication points.

The core algorithm of **i-UIP** as a clause reduction technique is simple. However, to make the algorithm a practical learning scheme that is compatible with modern SAT solvers, a number of issues need to be addressed. The rest of the section details the key optimizations and augmentations of **i-UIP** as a practical clause learning scheme.

**Algorithm 2** Control-**i-UIP**

**Require:** $C_1$ is a valid 1-UIP clause
**Require:** $t_{gap} \geq 0$ is a dynamically calculated gap threshold
1: **procedure** CONTROL-**i-UIP**$(C_1, t_{gap})$
2: $Gap \leftarrow |C_1| - LBD(C_1)$
3: **if** $Gap > t_{gap}$ **then**
4: $C_i \leftarrow$ **i-UIP**$(C_1, \phi)$
5: **I-UIP-Greedy**$(C_i\ , C_1)$ ▷ additional clause selection policy, see sec 3.3
6: **if** $|C_i| < |C_1|$ **then**
7: Succeed $\leftarrow$ Succeed + 1
8: **end if**
9: Attempted $\leftarrow$ Attempted + 1
10: **end if**
11: **end procedure**

### 3.1 Control i-UIP Learning

This simple and greedy **i-UIP** learning scheme can produce significantly smaller clause. However, when **i-UIP** does not reduce the clause size, the cost of the additional resolution steps will hurt the solver's performance. Since resolution cannot reduce a clause's LBD, The maximum size reduction from **i-UIP** is the difference between the clause's size and LBD, denote as the clause's gap value ($\text{Gap}(C_1) = |C_1| - \text{LBD}(C_1)$). For an 1-UIP clause with a small Gap, applying **i-UIP** is unlikely to achieve cost effective results. Therefore, we propose a heuristic based approach to to enable and disable **i-UIP** learning based on input clause's Gap.

Alg. 2 compares the input $C_1$'s Gap against a floating target threshold $t_{gap}$ (line 3). The threshold $t_{gap}$ represent the expected minimal Gap required for $C_1$ to achieve a predetermined success rate (80%) from performing **i-UIP** learning.

The gap threshold $t_{gap}$ is initialized to 0, and is updated at every restart based on **i-UIP**'s success rate from the previous restart interval. More specifically, the algorithm collects the statistics of the number of **i-UIP** learning attempted (line 9 in alg. 2) and the number of attempts succeed (line 7) for each restart interval, and use them to calculate the success rate. If the success rate is below 80, the threshold $t_{gap}$ is increased to restrict **i-UIP**for the next restart interval. On the other hand, the threshold is decreased to encourage more aggressive **i-UIP** learning for the next restart interval.

$$t_{gap} = \begin{cases} t_{gap} + 1 & \text{if } \frac{\text{Succeed}}{\text{Attempted}} < 0.8 \\ max(t_{gap} - 1, 0), & \text{otherwise} \end{cases}$$

### 3.2 Early stop i-UIP Learning

At any point of **i-UIP** learning, if the number of marked literals in $C_i$ (literals which are forced into the clause to preserve LBD) exceeds the input clause $C_1$'s

**Algorithm 3 i-UIP-Greedy**

**Require:** $C_i$ is a valid i-UIP clause
**Require:** $C_1$ is a valid 1-UIP clause
1: **procedure i-UIP-Greedy**$(C_i, C_1)$
2:     **if** $|C_i| < |C_1| \wedge (\text{AvgVarAct}(C_i) > \text{AvgVarAct}(C_1))$ **then**
3:         **return** $C_i$
4:     **else**
5:         **return** $C_1$
6:     **end if**
7: **end procedure**

Gap, we can abort **i-UIP** learning for $C_1$ because the size of the final $C_i$ is at least the size of $C_1$. The early stopping rule prevents solver from wasting time on traversing a large implication graph when **i-UIP** has already failed.

### 3.3 Greedy Active Clause Selection

Even though **i-UIP** learning can reduce the size of the learnt clause $C_i$, but it may introduce literals with low variable activity into $C_i$. Inactive literals prevents the clause from being asserted to force literal implication. Therefore, a practical clause learning scheme should consider both size and variable activity. We propose an optional extension **i-UIP-Greedy** to filter out inactive $C_1$.

After computing the $C_i$, **i-UIP-Greedy** compares both the size and the average variable activity for $C_1$ and $C_i$ (alg. 3 at line 2). The algorithm learns $C_i$ if the clause has smaller size and higher average variable activity.

### 3.4 Adjust Variable Activity

Two popular branching heuristics for modern SAT solvers are VSIDS and LBR. Both heuristics increase the variable activity for all literals involved in resolutions during 1-UIP learning. Since **i-UIP** extends 1-UIP with deeper resolutions against the trail, the variables activities for the fresh literals involved in the additional resolution steps need to be adjusted as well. We purpose two optional schemes for adjusting variable activities, **i-UIP-Inclusive** and **i-UIP-Exclusive**.

After learning $C_1$ from 1-UIP scheme, **i-UIP-Inclusive** collects all literals appear in the the **i-UIP** clause $C_i$, and increase their variable activity uniformly according to the current branching heuristic if the literals' variable activity have not yet been bumped during 1-UIP. The scheme does not bump variable activities for transient literals that are resolved away at non-conflicting level because, unlike transient literals at conflicting level, these literals cannot be re-asserted after the immediate backtracking. Notice that other post-analyze extensions such as Reason Side Rate (RSR) and Locality[] can be applied after applying **i-UIP-Inclusive** on $C_i$ so that no literal's variable activity is double bumped.

$$\forall l \in C_i \cdot NotBumped(var(l)) \implies bumpActivity(var(l))$$

**i-UIP-Exclusive** collects literals appear exclusively in $C_i$ and bump their variable activity uniformly. It also find all the literals in $C_1$ that are resolved away during **i-UIP**, and unbump their variable activity if they have been bumped during 1-UIP. The unbumped literals are no longer in the learned clause $C_i$, keep their variable activity bumped does not help solver to use $C_i$.

$$\forall l_i \in C_i \setminus C_1 \cdot bumpActivity(var(l_i))$$
$$\forall l_1 \in C_1 \setminus C_i \cdot unbumpActivity(var(l_1))$$

### 3.5 Integrate with Chronological Backtracking

In SAT solver with chronological backtracking, literals on the assertion trail $\phi$ are not always sorted by decision levels. This change imposes a challenge to **i-UIP** learning since the previous implementation relies on solver's ability to efficiently access all literals from any decision level in descending trail order (alg. 1 line 7).

To mitigate this challenge, we modify the solver to track the precise trail position of all asserted literals with a single vector. We then build a priority queue *lit_Order* to manage literal's resolution order. The queue *lit_Order* prioritizes literals with higher decision level, and it favors literal with higher trail position when decision levels are tied. The order of *lit_Order* represents the correct resolution order of **i-UIP** because 1) an asserted literal's decision level is the maximum level among all of its reasoning literals, and 2) an asserted literal always appears higher in the trail than its reasoning literal.

Alg. 4 is the pesudo-code implementation of the augmented **i-UIP** for chronological backtracking with the priority queue *lit_Order*. The algorithm first populates *lit_Order* with all literals in $C_1$ (line 4), and then continuously pops literals until the queue is empty (line 6). When a literal is resolved away, all of its reasoning literals are added into *lit_Order* if they are not already in the queue (line 13). The algorithm will always terminate because a literal cannot entered the queue twice (guaranteed by the properties of the trail order) and there are finite amount of literals on the trail.

## 4 Implementation and Experiments

### 4.1 Clause Reduction with i-UIP

To evaluate **i-UIP**'s effectiveness as a clause reduction technique, we implement **Pure-i-UIP** and **Min-i-UIP** on top of *MapleCOMSPS_LRB* [], the winner of SAT Race 2015 application track. We than compare the performance of the baseline *MapleCOMSPS_LRB* with Maple-**Pure-i-UIP** and Maple-**Min-i-UIP** on the full set of benchmarks from SAT RACE 2019 main track.

The benchmark contains 400 instances divided into two groups of 200, new and old, representing historical instances and fresh instances in the 2019 race,

**Algorithm 4 i-UIP**-CB

**Require:** $C_1$ is a valid 1-UIP clause
**Require:** $\phi$ is a valid assertion trail
**Require:** $lit\_Order$ is a priority queue
**procedure i-UIP**-CB($C_1, \phi, lit\_Order$)
    ...
    $C_i \leftarrow C_1$ ▷ initialize i-UIP clause
    forall $l \in C_1 \cdot$ Enqueu($lit\_Order, l$)
    DecisionLvs $\leftarrow \{$ level($l$) $\mid l \in C_1 \}$
    **while** $lit\_Order \neq \emptyset$ **do**
        $p \leftarrow$ dequeu($lit\_Order, l$)
        **if** $\exists q \in$ Reason($\neg p, \phi$) $\cdot$ level($q$) $\notin$ DecisionLvs
$\vee p$ is the last reminaing lit in its decision level **then**
            Pass
        **else**
            $C_i \leftarrow (C_1) \bowtie$ Reason($q, \phi$)
            forall $l \in$ Reason($q, \phi$) $\cdot$ Enqueu($lit\_Order, l$)
        **end if**
    **end while**
    ...
**end procedure**

respectively. I partition the old group instances into six partitions of size 30 and one partition of size 20. Each partition is then assigned to a Intel CPU node with 16 cores (2 sockets 8 cores and 1 thread) and 96649 MB memory. The new group is partitioned based on their contributor (e.g. Heule contributed 22 matrix multiplication instances), and each partition is assigned to a aforementioned CPU node. To speed up the experiment, we allow a CPU node to solver at most seven instances concurrently.

Beside solved instances count and PAR-2 score, we additionally measure the average clause length and clause reduction ratio for each instances. For Maple-**Pure-i-UIP** and Maple-**Min-i-UIP**, we also captures the **i-UIP** learning attempted rate and success rate.

| Solver | # solved | PAR-2 | Clause Size | Cl Reduction% |
|---|---|---|---|---|
| *MapleCOMSPS_LRB* | 221 | 5018.89 | 62.6 | 36.53% |
| Maple-**Pure-i-UIP** | 226 | 4920.04 | 49.6 | 41.6% |
| Maple-**Min-i-UIP** | 226 | 4890.67 | 45.2 | 47.8% |

Fig. 1: Benchmark results of *MapleCOMSPS_LRB* , Maple-**Pure-i-UIP** and Maple-**Min-i-UIP** on SAT2019 race main track.

Fig. 1 shows that both version of **i-UIP** solved five more instances than the baseline solver with lower PAR-2 scores. **Min-i-UIP** has marginally lower

PAR-2 score than **Pure-i-UIP**. Both **Pure-i-UIP** and **Min-i-UIP** produce clause with significantly smaller size than 1-UIP by 27.7% and 20.7%, respectively. Fig. 2 shows the probability density distribution (PDF) of the average clause length from **Min-i-UIP** relative to 1-UIP. **Min-i-UIP** learning produces shorter clauses for 88.25% instances, and average relative reduction from 1-UIP is 18.685%.Fig. 3 compares the absolute average clause size from **Min-i-UIP** and 1-UIP, and it shows that **Min-i-UIP** in general produces smaller clauses, and the size reduction is more significant for instances with large average 1-UIP clause size.

We also looked at the 14 instances solved by **Min-i-UIP** but not by 1-UIP. **Min-i-UIP** produces smaller clauses for all of them with average relative reduction of 22% and maximum 77% (30 vs 135). Seven out of 14 instances has size relative reduction over 30%. For the 9 instances solved by 1-UIP but not by **Min-i-UIP**, **Min-i-UIP** only produce smaller clause by 33% and with average relative reduction of 3.3$.

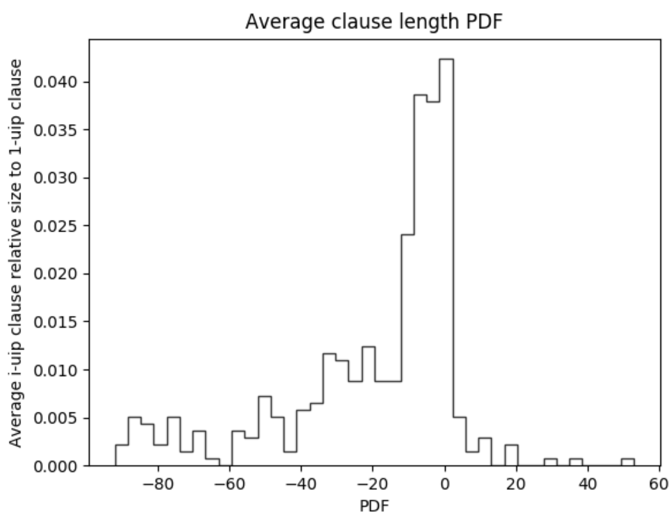


Fig. 2: Average clause (relative to 1-UIP clauses) size distribution. X axis indicates the relative size difference, and Y axis indicates the PDF.

**Min-i-UIP** outperformed **Pure-i-UIP** in both PAR-2 score and clause size. This results agrees with our observation in Fig. 4: **Min-i-UIP** attempted **i-UIP** learning more frequently, and it is more likely to succeed. Remark that the success of **i-UIP** learning is determined by the size of the learned i-UIP clause $C_i$, and the **i-UIP** learning frequency is also indirectly controlled by **i-UIP**'s success rate from the previous restart interval. The results indicates **Min-i-UIP**

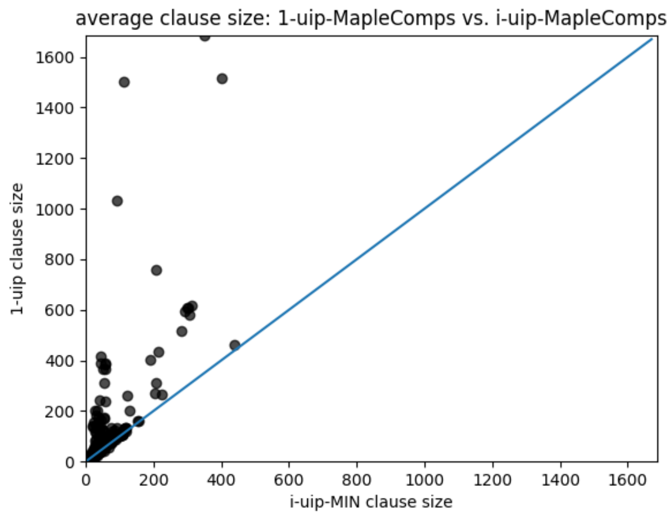


Fig. 3: Average clause size comparison plot. Each point in the plot represents an benchmark instance. X and Y axis shows the clause length from Maple-**Min-i-UIP** and *MapleCOMSPS_LRB*, respectively

shortened $C_i$'s size through further minimization of $C_i$ at non-unique implication decision level.

| Solver | **i-UIP** attempt rate | **i-UIP** success rate |
|---|---|---|
| Maple-**Pure-i-UIP** | 16.1% | 43.4% |
| Maple-**Min-i-UIP** | 28.8% | 59.3% |

Fig. 4: Compare **Pure-i-UIP** and **Min-i-UIP** i-uip attempt rate and success rate. **Min-i-UIP** scheme attempted **i-UIP** more frequently, and it is more likely to successfully produce smaller $C_i$ clause .

A solver produce smaller clauses can construct smaller proofs. For UNSAT instances, we additionally measure their DRUP[] proof checking time as well as the size of the optimized DRUP proof. We used DART-trim [] with 5000 timeout to check and optimize DRUP proofs.

Fig. 5 shows that the optimized proof construct by **Min-i-UIP** and **Pure-i-UIP** are significantly smaller than 1-UIP proofs. The relative proof size reduction rougly correlates to the average clause size reduction. Fig. 6 shows the absolute proof size comparison results.

| Solver | optimized proof size (MB) | relative reduction size |
|---|---|---|
| *MapleCOMSPS_LRB* | 613.9 | 0 |
| Maple-**Pure-i-UIP** | 487.2 | 6.90% |
| Maple-**Min-i-UIP** | 413.2 | 17.18% |

Fig. 5: Optimized UNSAT proof comparison for 1-UIP **Pure-i-UIP** and **Min-i-UIP**. Optimized proof size measures the average absolute proof size in MB, and relative reduction size measures the average relative reduction for all UNSAT instances.

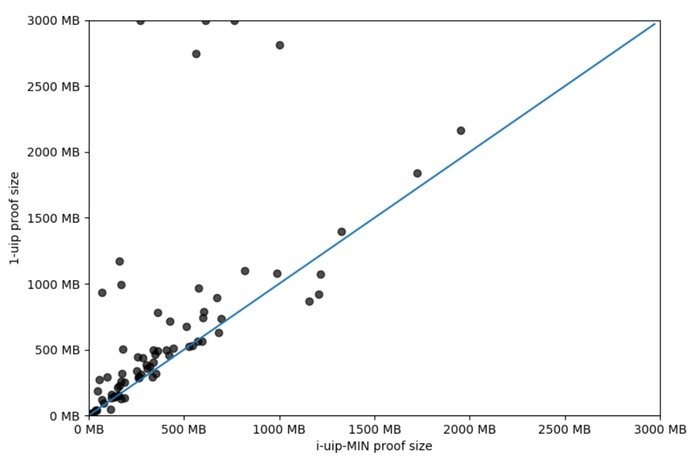


Fig. 6: Average optimized proof size between 1-uip and **Min-i-UIP**.

### 4.2 i-UIP as a Practical Learning Scheme

To evaluate **i-UIP**'s effectiveness as a clause learning scheme, we implement **Min-i-UIP** on *MapleCOMSPS_LRB* with the extensions mentioned in section **??**. We evaluated four different configurations (i-UIP, i-UIP-Greedy, i-UIP-Inclusive, and i-UIP-Exclusive) of **i-UIP** and 1-UIP learning on the SAT Race 2019 main track benchmark and report each configuration's solved instances, PAR-2 score and average clause size.

Fig. 7 summarizes the result of the experiment. Learning scheme i-UIP-greedy solves the same amount of instance (226) as i-UIP with less PAR-2 score. The inclusive activity adjustment solves the most SAT instances (138) and the least UNSAT instances (87). The exclusive activity adjustment scheme produces the shortest average clause size, but solved the second least instances, one more instance than the baseline.

| Solver | # solved (SAT, UNSAT) | PAR-2 | Avg clause Size |
|---|---|---|---|
| 1-UIP | 221 (132, 89) | 5018.89 | 62.6 |
| i-UIP | **226** (135, **91**) | 4890.67 | 45.2 |
| i-UIP-greedy | **226** (135, **91**) | **4866.94** | 47.7 |
| i-UIP-active-Inclusive | 225 (**138**, 87) | 4958.49 | 52..12 |
| i-UIP-active-Exclusive | 223 (134, 89) | 5015.23 | **43.2** |

Fig. 7: Benchmark results of 1-UIP (*MapleCOMSPS_LRB*), i-UIP(**Min-i-UIP**), i-UIP-Greedy, i-UIP-Inclusive, and i-UIP-Exclusive on SAT2019 race main track.

Inserted a table here, and graphs and analysis

### 4.3 i-UIP on Modern SAT solvers

To validate **i-UIP** as a generalizable learning scheme on modern SAT solvers, we re-implement **i-UIP** on *MapleLCMDist* [], *MapleLCMDiscChronoBT-DL-v3* [] and *expMaple_CM_GCBumpOnlyLRB*. The first two solvers are the winner of 2017 and 2019 SAT race, respectively. *expMaple_CM_GCBumpOnlyLRB* is a top ten solver from 2019 SAT race which uses random walk simulation to help branching. For each solver, we compare the base 1-UIP learning scheme against top two **i-UIP** configurations, **i-UIP-Greedy** and **i-UIP-Inclusive**, on the SAT Race 2019 main track benchmark. We report solved instances and PAR-2 score.

Inserted a table here, and graphs and analysis

## References

1. LNCS Homepage, http://www.springer.com/lncs. Last accessed 4 Oct 2017