Core Challenge 2022
Solver and Graph Descriptions

Version 0.1

Edited by

Takehide Soh
Kobe University, Japan

Yoshio Okamoto
The University of Electro-Communications, Japan

Takehiro Ito
Tohoku University, Japan

Contents

1 Finding Shortest Reconfigurations Sequences of Independent Sets **3**

2 (PARIS) Planning Algorithms for Reconfiguring Independent Sets **15**

3 Greedy BMC Solver for the Independent Set Reconfiguration Problem **23**

4 ISR Solver Track Documentation **25**

5 recongo: ASP-based Combinatorial Reconfiguration Problem Solver **26**

6 ReconfAIGERation entering Core Challenge 2022 **27**

7 A decision diagram-based solver for the independent set reconfiguration problem **30**

8 Graph track description **31**

9 Every Reconfiguration Starts with a First Step **34**

10 A Series of Graphs With Exponentially Growing Reconfigurations Sequences of Independent Sets **37**

11 Every House on the Block: A generalized solution to creating ISR instances of large plan length **44**

12 The Documentation of Graph Track in Core Challenge **49**

Finding Shortest Reconfigurations Sequences of Independent Sets

Volker Turau and Christoph Weyer

Institute of Telematics, Hamburg University of Technology
Hamburg, Germany
turau@tuhh.de

1 Algorithm

The problem at hand is a classical problem in graph theory: find a path between two given vertices in an undirected graph such that the sum of the weights of its constituent edges is minimized. Since in our case all edge weights are equal to 1 this is equivalent to finding the path with fewest edges. There is a myriad of algorithms available for this problem. The challenge in this case is that the graph (a.k.a. the reconfiguration graph) as is only defined implicitly, i.e., the vertices and edges are not explicitly given but rather determined algorithmicly from the input. In addition this implicit graph is rather large, too large to fit into the main memory of current computers. This situation is very common in many search problems. Hence, search algorithms such as A* are good candidates to solve the given problem. The disadvantage of these algorithms is that they keep all generated vertices in memory. For example A* keeps two lists: *open* and *closed* vertices and all vertices touched before finding a solution end up in one of the two lists. Thus, for graphs with more than 10^{10} vertices this requires at 100 GB to 1 T GB of RAM. This outstrips current off-the-shelf hardware. Therefore memory-bounded heuristic searches such as SMA* or IDA* are a better choice. We use Iterative Deepening A* (IDA*) and adapt it to our needs.

IDA* roughly works as follows: In each iteration it performs a limited horizon depth-first search. The horizon is defined using a heuristic estimate of the length of the path from the current vertex to the target vertex. The horizon is the sum of the current distance and the heuristic estimate. The horizon is increased at the end of each iteration by the smallest value that exceed the last horizon. There is a trade of between the quality of a heuristic and the complexity of evaluating the heuristic. Since for large search problems the heuristic is evaluated extremely often it is necessary to choose a simple heuristic that can be preferably computed in constant time. Since in our case the vertices of the implicit graph correspond to independent sets of the input graph, we used the

following simple heuristic to estimate the distance from the current independent set C to the target independent set T: $h(C) := |C \setminus T|$, i.e., the node of vertices that are not yet in the target set. This heuristic is easy to compute. Also it is *consistent*, this guarantees that IDA* finds the shortest path.

The downside of informed search algorithms is that if there is no solution – i.e., start and target vertex are in different connected components – they have to access every vertex of the connected component of the graph that contains the start vertex at least once. This problem is aggravated by the fact, that IDA* increases its horizon gradually. So in case no solution exists the number of operations roughly grows exponentially with the diameter of the graph. Thus, in this case a simpler approach is required. We have used a variant of breadth-first search (BFS) for this purpose, that does not require to explicitly store the visited vertices and evaluates no heuristic.

Since it is not known upfront whether a solution exits we have decided to use both approaches concurrently. After initializing the data structures from the input files we launch two threads. The first one executes our version of the breadth-first search while the second executes our variant of IDA*. Whichever thread first finds a shortest path between start and target vertex or finds out that these vertices lie in in different connected components of the implicit graph ends the program and outputs a solution.

The main challenge in solving shortest path problems for large graphs is to use a data structure that allows to quickly determine the neighbors of a vertex in the implicit graph and to quickly evaluate the heuristic. Since vertices in the implicit graph correspond to independent sets we used an adjacency matrix to represent the input graph. Each row of this matrix consists of n bits (n is the number of vertices of the input graph). Thus, a row is represent as $\lceil n/64 \rceil$ words of size 64. The reason for this choice is that current computers have a word size of 64 bits. A vertex of the implicit graph – an independent set of the input graph – is thus also represented by a bit field consisting of $\lceil n/64 \rceil$ words. Thus, given that X is an independent set, to check whether $X \setminus \{x\} \cup \{y\}$ is also independent can be decided by at most $\lceil n/64 \rceil$ XOR operations. We can update independent sets in constant time. Similarly, we can for a given independent set compute its neighboring independent sets in time $O(n)$ time. Moreover, very efficient low level functions for this type of representation are available, e.g., *__builtin_ffsll* for computing the index of the first bit set. At no time we have to iterate over the neighborhood of a vertex.

To cope with the large number of vertices we used different techniques. In BFS we represented the vertices of every layer in a separate AVL tree. We maintain at any time only the top two layers of the BFS tree. The predecessor relation is maintained with reference counters. At the completion of each layer, lower layers are thinned out by recursively removing vertices with no successor, i.e., reference counter 0. Vertices in the AVL trees correspond to independent sets. In order to speed up the search in such trees we additionally stored a 64 Bit hash value with each independent set. Thus, first we compared the hash values and only if these coincided we needed to explicitly compare the independent sets. Therefore, in most cases a comparison only required one operation.

We implemented a variant of IDA*. To speed up this algorithm we maintained a compact history between individual iterations. This history includes information at what distance from the start vertex a vertex appeared. If it reappears at the same or a larger distance it can be skipped. At the beginning of a new iteration the history was *lifted*, i.e., the distance of each vertex is incremented by 1, so that we search beyond the previous horizon. This helps to considerably reduce the number of vertices that had to be revisited. Also, since this history is not required for the algorithm to work correctly, we can at any time close the history to limit the usage of RAM. The history is implemented as hash table using open addressing, in particular we used linear probing, the hash function *MurmurHash2* yielding hash values of size 64 bit, and a load factor of 5/8. We also designed the data structures such as AVL trees in a way that allows to efficiently allocate and deallocate the required memory.

The algorithm is to a large degree a general search algorithm. It can use the TS (token sliding) or the TJ (token jumping) model. Some of the problem specific aspects are as follows.

- The estimate function $h(C)$, it can be computed in constant time given the value of the previous state.
- The successors of a vertex in the search graph can be found very efficiently by maintaining for each node the number of neighbors that are in the current independent set. There are only two cases. If a node has no such neighbor, then every token can be placed on it. If it has a single such neighbor then the token of this neighbor can be placed on it.
- Representing the graph as a bitset allowed to implement many basic graph operations with logical operators such as *and, or*, and *xor* on words of length 64 bit.

A simple extension of the algorithm would be to run two more threads that start to search from the target independent set. For the graphs in the benchmark that did not bring any benefit, because they seem to be *symmetric* with respect to start and target. We do not know whether this property holds in general, properly not.

2 Implementation Details

The algorithm is implemented in the programming language C, using c99. As a compiler we used *clang* with options *-Ofast -march=native -mtune=native*. The source code consists of seven modules, in total about 50 KiB of code. The threads are implemented using *pthread*. We do not use any third party software.

3 Computation Environment

We run our benchmarks on the following configuration. We used an Ubuntu 20.04 virtual machine with 4 logical CPUs and 60 GiB RAM. Our program uses

Dat File	n	m	Solution	Time
grid004x004_01.dat	16	24	No	00:00:00.07
grid004x004_02.dat	16	24	No	00:00:00.05
grid004x004_03.dat	16	24	7	00:00:00.05
grid004x004_04.dat	16	24	8	00:00:00.05
grid004x004_05.dat	16	24	No	00:00:00.07
hc-power-11_01.dat	21	28	21	00:00:00.06
hc-power-12_01.dat	36	51	69	00:00:00.05
hc-square-01_01.dat	14	18	12	00:00:00.05
hc-square-02_01.dat	24	33	30	00:00:00.05
hc-toyno-01_01.dat	6	9	No	00:00:00.05
hc-toyyes-01_01.dat	7	7	3	00:00:00.05

Table 1: Solutions for first benchmark

only 2 CPUs. The server uses Proxmox as virtualization environment. The server has an AMD Ryzen9 3900X CPU with 12 cores at 3.80 GHz. For storage we use NVMe SSDs.

4 Solutions

In the following tables we summarize the outcome of our program. The corresponding output files can be found in die `solutions` directory. The column solution specifies if no path can be found or the number of steps for the shortest path. The time column is the measured elapsed time from the Unix tool `time` as `HH:MM:SS.mm`. If a program runs longer than one hour no milliseconds are given.

When we found no solution for the given problem due to time constraints a — is shown.

Dat File	n	m	Solution	Time
grid010x010_01_6141.dat	100	180	No	00:00:00.05
grid010x010_02_0495.dat	100	180	No	00:00:00.05
grid010x010_03_6844.dat	100	180	No	00:00:00.05
grid010x010_04_4957.dat	100	180	No	00:00:00.08
grid020x020_01_9275.dat	400	760	No	00:00:01.40
grid020x020_02_7550.dat	400	760	No	00:00:00.07
grid020x020_03_3902.dat	400	760	No	00:00:01.44
grid020x020_04_3283.dat	400	760	No	00:00:00.05
grid030x030_01_6931.dat	900	1740	No	00:00:00.05
grid030x030_02_7112.dat	900	1740	No	00:00:21.81
grid030x030_03_4248.dat	900	1740	No	00:00:00.05
grid030x030_04_2988.dat	900	1740	No	00:00:22.17
grid040x040_01_6703.dat	1600	3120	No	00:00:00.13
grid040x040_02_3552.dat	1600	3120	No	00:00:00.09
grid040x040_03_0746.dat	1600	3120	No	00:02:29.34
grid040x040_04_3680.dat	1600	3120	No	00:02:35.90
grid050x050_01_8271.dat	2500	4900	No	00:00:00.31
grid050x050_02_8314.dat	2500	4900	No	00:14:18.34
grid050x050_03_4298.dat	2500	4900	No	00:00:00.29
grid050x050_04_8232.dat	2500	4900	No	00:14:16.52
grid060x060_01_5912.dat	3600	7080	No	00:00:00.62
grid060x060_02_4571.dat	3600	7080	No	00:57:59.71
grid060x060_03_2183.dat	3600	7080	No	00:57:30.17
grid060x060_04_3263.dat	3600	7080	No	00:00:00.63
grid070x070_01_6558.dat	4900	9660	No	00:00:01.41
grid070x070_02_7903.dat	4900	9660	No	02:43:26
grid070x070_03_4857.dat	4900	9660	No	02:45:27
grid070x070_04_3525.dat	4900	9660	No	00:00:01.35
grid080x080_01_9321.dat	6400	12640	No	00:00:02.76
grid080x080_02_8623.dat	6400	12640	No	06:50:57
grid080x080_03_8653.dat	6400	12640	No	00:00:02.77
grid080x080_04_8592.dat	6400	12640	No	06:55:10
grid090x090_01_0710.dat	8100	16020	No	18:55:31
grid090x090_02_9631.dat	8100	16020	No	00:00:07.58
grid090x090_03_8778.dat	8100	16020	No	19:37:18
grid090x090_04_4969.dat	8100	16020	No	00:00:07.56
grid100x100_01_9072.dat	10000	19800	No	38:47:16
grid100x100_02_7191.dat	10000	19800	No	00:00:11.73
grid100x100_03_4887.dat	10000	19800	No	39:08:22
grid100x100_04_0664.dat	10000	19800	No	00:00:11.96
grid200x200_01_8265.dat	40000	79600	—	—
grid200x200_02_2458.dat	40000	79600	No	00:11:13.24
grid200x200_03_0743.dat	40000	79600	No	00:11:09.19
grid200x200_04_2816.dat	40000	79600	—	—

Table 2: Solutions for grid benchmarks

Dat File	n	m	Solution	Time
queen008x008_01_4136.dat	64	728	9	00:00:00.13
queen008x008_02_8804.dat	64	728	11	00:00:00.05
queen008x008_03_8723.dat	64	728	11	00:00:00.05
queen008x008_04_4441.dat	64	728	12	00:00:00.05
queen010x010_01_0612.dat	100	1470	10	00:00:00.05
queen010x010_02_5832.dat	100	1470	11	00:00:00.05
queen010x010_03_7635.dat	100	1470	13	00:00:00.05
queen010x010_04_2943.dat	100	1470	14	00:00:00.05
queen020x020_01_7075.dat	400	12540	24	00:00:00.57
queen020x020_02_4815.dat	400	12540	22	00:00:00.31
queen020x020_03_8667.dat	400	12540	21	00:00:00.38
queen020x020_04_0961.dat	400	12540	18	00:00:00.25
queen030x030_01_3761.dat	900	43210	15	00:00:00.13
queen030x030_02_4963.dat	900	43210	26	00:00:03.61
queen030x030_03_7394.dat	900	43210	26	00:00:02.41
queen030x030_04_9144.dat	900	43210	24	00:00:00.66
queen040x040_01_6291.dat	1600	103480	13	00:00:00.08
queen040x040_02_9487.dat	1600	103480	14	00:00:00.14
queen040x040_03_7037.dat	1600	103480	14	00:00:00.12
queen040x040_04_8369.dat	1600	103480	17	00:00:00.14
queen050x050_01_6006.dat	2500	203350	13	00:00:00.12
queen050x050_02_6166.dat	2500	203350	16	00:00:00.12
queen050x050_03_0562.dat	2500	203350	6	00:00:00.08
queen050x050_04_9047.dat	2500	203350	12	00:00:00.10
queen060x060_01_8336.dat	3600	352820	5	00:00:00.12
queen060x060_02_6710.dat	3600	352820	9	00:00:00.12
queen060x060_03_1394.dat	3600	352820	9	00:00:00.10
queen060x060_04_3799.dat	3600	352820	9	00:00:00.11
queen070x070_01_1828.dat	4900	561890	38	04:21:08
queen070x070_02_8683.dat	4900	561890	40	00:38:16.40
queen070x070_03_1545.dat	4900	561890	34	00:01:23.21
queen070x070_04_7426.dat	4900	561890	33	00:01:06.70
queen080x080_01_0297.dat	6400	840560	25	00:00:10.44
queen080x080_02_2368.dat	6400	840560	31	00:05:34.91
queen080x080_03_0355.dat	6400	840560	—	—
queen080x080_04_4954.dat	6400	840560	36	00:10:46.40
queen090x090_01_8978.dat	8100	1198830	—	—
queen090x090_02_0808.dat	8100	1198830	41	02:06:41
queen090x090_03_9570.dat	8100	1198830	—	—
queen090x090_04_1188.dat	8100	1198830	38	00:05:02.33
queen100x100_01_3955.dat	10000	1646700	—	—
queen100x100_02_1699.dat	10000	1646700	6	00:00:00.38
queen100x100_03_6670.dat	10000	1646700	—	—
queen100x100_04_0618.dat	10000	1646700	—	—
queen200x200_01_0761.dat	40000	13253400	4	00:00:02.09
queen200x200_02_3057.dat	40000	13253400	29	00:00:10.10
queen200x200_03_0327.dat	40000	13253400	29	00:00:10.08
queen200x200_04_4796.dat	40000	13253400	27	00:00:03.06

Table 3: Solutions for queen benchmarks

Dat File	n	m	Solution	Time
hc-square-004-002_01.dat	44	63	90	00:00:00.05
hc-square-005-002_01.dat	54	78	132	00:00:00.05
hc-square-006-002_01.dat	64	93	182	00:00:00.07
hc-square-007-002_01.dat	74	108	240	00:00:00.49
hc-square-008-002_01.dat	84	123	306	00:00:03.39
hc-square-009-002_01.dat	94	138	380	00:00:25.28
hc-square-010-002_01.dat	104	153	462	00:03:04.50
hc-square-011-002_01.dat	114	168	552	00:24:20.14
hc-square-012-002_01.dat	124	183	650	03:06:18
hc-square-013-002_01.dat	134	198	756	23:08:27
hc-square-014-002_01.dat	144	213	—	—
hc-square-015-002_01.dat	154	228	—	—
hc-square-016-002_01.dat	164	243	—	—
hc-square-017-002_01.dat	174	258	—	—
hc-square-018-002_01.dat	184	273	—	—
hc-square-019-002_01.dat	194	288	—	—
hc-square-020-002_01.dat	204	303	—	—

Table 4: Solutions for square benchmarks

Dat File	n	m	Solution	Time
hc-power-004-002_01.dat	64	95	359	00:00:00.07
hc-power-005-002_01.dat	79	118	779	00:00:00.21
hc-power-006-002_01.dat	94	141	1631	00:00:02.24
hc-power-007-002_01.dat	109	164	3347	00:00:25.03
hc-power-008-002_01.dat	124	187	6791	00:04:39.50
hc-power-009-002_01.dat	139	210	13691	00:58:42.48
hc-power-010-002_01.dat	154	233	27503	11:00:38
hc-power-011-002_01.dat	169	256	—	—
hc-power-012-002_01.dat	184	279	—	—
hc-power-013-002_01.dat	199	302	—	—
hc-power-014-002_01.dat	214	325	—	—
hc-power-015-002_01.dat	229	348	—	—
hc-power-016-002_01.dat	244	371	—	—
hc-power-017-002_01.dat	259	394	—	—
hc-power-018-002_01.dat	274	417	—	—
hc-power-019-002_01.dat	289	440	—	—
hc-power-020-002_01.dat	304	463	—	—

Table 5: Solutions for power benchmarks

Dat File	n	m	Solution	Time
sp001_01.dat	13	66	11	00:00:00.05
sp002_01.dat	26	150	33	00:00:00.05
sp003_01.dat	39	234	77	00:00:00.05
sp004_01.dat	52	318	165	00:00:00.05
sp005_01.dat	65	402	341	00:00:00.05
sp006_01.dat	78	486	693	00:00:00.20
sp007_01.dat	91	570	1397	00:00:01.77
sp008_01.dat	104	654	2805	00:00:15.77
sp009_01.dat	117	738	5621	00:02:27.31
sp010_01.dat	130	822	11253	00:24:02.35
sp011_01.dat	143	906	22517	04:01:44
sp012_01.dat	156	990	—	—
sp013_01.dat	169	1074	—	—
sp014_01.dat	182	1158	—	—
sp015_01.dat	195	1242	—	—
sp016_01.dat	208	1326	—	—
sp017_01.dat	221	1410	—	—
sp018_01.dat	234	1494	—	—
sp019_01.dat	247	1578	—	—
sp020_01.dat	260	1662	—	—
sp021_01.dat	273	1746	—	—
sp022_01.dat	286	1830	—	—
sp023_01.dat	299	1914	—	—
sp024_01.dat	312	1998	—	—
sp025_01.dat	325	2082	—	—
sp026_01.dat	338	2166	—	—
sp027_01.dat	351	2250	—	—
sp028_01.dat	364	2334	—	—
sp029_01.dat	377	2418	—	—
sp030_01.dat	390	2502	—	—

Table 6: Solutions for SP benchmarks

Dat File	n	m	Solution	Time
1-FullIns_3_01.dat	30	100	1	00:00:00.06
1-FullIns_3_02.dat	30	100	3	00:00:00.06
1-FullIns_4_01.dat	93	593	1	00:00:00.05
1-FullIns_4_02.dat	93	593	1	00:00:00.05
1-FullIns_5_01.dat	282	3247	1	00:00:00.07
1-FullIns_5_02.dat	282	3247	1	00:00:00.06
1-Insertions_4_01.dat	67	232	3	00:00:00.05
1-Insertions_4_02.dat	67	232	2	00:00:00.05
1-Insertions_5_01.dat	202	1227	2	00:00:00.06
1-Insertions_5_02.dat	202	1227	2	00:00:00.05
1-Insertions_6_01.dat	607	6337	1	00:00:00.07
1-Insertions_6_02.dat	607	6337	3	00:00:00.05
2-FullIns_3_01.dat	52	201	2	00:00:00.05
2-FullIns_3_02.dat	52	201	2	00:00:00.05
2-FullIns_4_01.dat	212	1621	1	00:00:00.05
2-FullIns_4_02.dat	212	1621	1	00:00:00.05
2-FullIns_5_01.dat	852	12201	1	00:00:00.06
2-FullIns_5_02.dat	852	12201	1	00:00:00.05
2-Insertions_3_01.dat	37	72	3	00:00:00.06
2-Insertions_3_02.dat	37	72	2	00:00:00.05
2-Insertions_4_01.dat	149	541	1	00:00:00.05
2-Insertions_4_02.dat	149	541	1	00:00:00.05
2-Insertions_5_01.dat	597	3936	1	00:00:00.06
2-Insertions_5_02.dat	597	3936	1	00:00:00.05
3-FullIns_3_01.dat	80	346	2	00:00:00.05
3-FullIns_3_02.dat	80	346	4	00:00:00.06
3-FullIns_4_01.dat	405	3524	2	00:00:00.05
3-FullIns_4_02.dat	405	3524	3	00:00:00.05
3-FullIns_5_01.dat	2030	33751	1	00:00:00.06
3-FullIns_5_02.dat	2030	33751	5	00:00:00.06
3-Insertions_3_01.dat	56	110	4	00:00:00.05
3-Insertions_3_02.dat	56	110	2	00:00:00.05
3-Insertions_4_01.dat	281	1046	2	00:00:00.05
3-Insertions_4_02.dat	281	1046	5	00:00:00.05
3-Insertions_5_01.dat	1406	9695	1	00:00:00.08
3-Insertions_5_02.dat	1406	9695	1	00:00:00.05
4-FullIns_3_01.dat	114	541	1	00:00:00.05
4-FullIns_3_02.dat	114	541	1	00:00:00.05
4-FullIns_4_01.dat	690	6650	1	00:00:00.05
4-FullIns_4_02.dat	690	6650	2	00:00:00.05
4-FullIns_5_01.dat	4146	77305	1	00:00:00.07
4-FullIns_5_02.dat	4146	77305	1	00:00:00.07
4-Insertions_3_01.dat	79	156	1	00:00:00.05
4-Insertions_3_02.dat	79	156	3	00:00:00.05
4-Insertions_4_01.dat	475	1795	1	00:00:00.05
4-Insertions_4_02.dat	475	1795	1	00:00:00.05
5-FullIns_3_01.dat	154	792	1	00:00:00.05
5-FullIns_3_02.dat	154	792	5	00:00:00.05
5-FullIns_4_01.dat	1085	11395	1	00:00:00.07
5-FullIns_4_02.dat	1085	11395	6	00:00:00.05

Table 7: Solutions for color benchmarks (part 1)

Dat File	n	m	Solution	Time
DSJC1000.1_01.dat	1000	49629	1	00:00:00.06
DSJC1000.1_02.dat	1000	49629	3	00:00:00.06
DSJC1000.5_01.dat	1000	249826	6	00:00:00.10
DSJC1000.5_02.dat	1000	249826	2	00:00:00.08
DSJC1000.9_01.dat	1000	449449	2	00:00:00.11
DSJC1000.9_02.dat	1000	449449	2	00:00:00.11
DSJC125.1_01.dat	125	736	3	00:00:00.06
DSJC125.1_02.dat	125	736	5	00:00:00.05
DSJC125.5_01.dat	125	3891	4	00:00:00.06
DSJC125.5_02.dat	125	3891	2	00:00:00.05
DSJC125.9_01.dat	125	6961	1	00:00:00.05
DSJC125.9_02.dat	125	6961	8	00:00:00.05
DSJC250.1_01.dat	250	3218	1	00:00:00.05
DSJC250.1_02.dat	250	3218	1	00:00:00.05
DSJC250.5_01.dat	250	15668	1	00:00:00.05
DSJC250.5_02.dat	250	15668	5	00:00:00.05
DSJC250.9_01.dat	250	27897	5	00:00:00.22
DSJC250.9_02.dat	250	27897	5	00:00:00.05
DSJC500.1_01.dat	500	12458	3	00:00:00.05
DSJC500.1_02.dat	500	12458	6	00:00:00.05
DSJC500.5_01.dat	500	62624	1	00:00:00.07
DSJC500.5_02.dat	500	62624	1	00:00:00.06
DSJC500.9_01.dat	500	112437	6	00:00:00.08
DSJC500.9_02.dat	500	112437	9	00:00:00.06
DSJR500.1_01.dat	500	3555	8	00:00:00.05
DSJR500.1_02.dat	500	3555	13	00:00:00.05
DSJR500.1c_01.dat	500	121275	1	00:00:00.10
DSJR500.1c_02.dat	500	121275	4	00:00:00.06
DSJR500.5_01.dat	500	58862	2	00:00:00.06
DSJR500.5_02.dat	500	58862	6	00:00:00.07
anna_01.dat	138	493	5	00:00:00.05
anna_02.dat	138	493	13	00:00:00.05
ash331GPIA_01.dat	662	4181	1	00:00:00.05
ash331GPIA_02.dat	662	4181	1	00:00:00.06
ash608GPIA_01.dat	1216	7844	1	00:00:00.05
ash608GPIA_02.dat	1216	7844	1	00:00:00.05
ash958GPIA_01.dat	1916	12506	1	00:00:00.08
ash958GPIA_02.dat	1916	12506	1	00:00:00.06
david_01.dat	87	406	6	00:00:00.05
david_02.dat	87	406	9	00:00:00.05
games120_01.dat	120	638	7	00:00:00.05
games120_02.dat	120	638	13	00:00:00.05
huck_01.dat	74	301	7	00:00:00.05
huck_02.dat	74	301	7	00:00:00.05
latin_square_10_01.dat	900	307350	1	00:00:00.28
latin_square_10_02.dat	900	307350	1	00:00:00.09

Table 8: Solutions for color benchmarks (part 2)

Dat File	n	m	Solution	Time
le450_15a_01.dat	450	8168	3	00:00:00.06
le450_15a_02.dat	450	8168	3	00:00:00.05
le450_15b_01.dat	450	8169	2	00:00:00.05
le450_15b_02.dat	450	8169	5	00:00:00.05
le450_15c_01.dat	450	16680	2	00:00:00.05
le450_15c_02.dat	450	16680	3	00:00:00.05
le450_15d_01.dat	450	16750	2	00:00:00.05
le450_15d_02.dat	450	16750	6	00:00:00.05
le450_25a_01.dat	450	8260	3	00:00:00.07
le450_25a_02.dat	450	8260	12	00:00:00.05
le450_25b_01.dat	450	8263	6	00:00:00.05
le450_25b_02.dat	450	8263	10	00:00:00.05
le450_25c_01.dat	450	17343	3	00:00:00.07
le450_25c_02.dat	450	17343	5	00:00:00.05
le450_25d_01.dat	450	17425	3	00:00:00.05
le450_25d_02.dat	450	17425	2	00:00:00.05
le450_5a_01.dat	450	5714	1	00:00:00.05
le450_5a_02.dat	450	5714	1	00:00:00.05
le450_5b_01.dat	450	5734	1	00:00:00.06
le450_5b_02.dat	450	5734	2	00:00:00.05
le450_5c_01.dat	450	9803	1	00:00:00.05
le450_5c_02.dat	450	9803	1	00:00:00.05
le450_5d_01.dat	450	9757	1	00:00:00.05
le450_5d_02.dat	450	9757	1	00:00:00.05
miles1000_01.dat	128	3216	3	00:00:00.05
miles1000_02.dat	128	3216	5	00:00:00.05
miles1500_01.dat	128	5198	2	00:00:00.05
miles1500_02.dat	128	5198	3	00:00:00.05
miles500_01.dat	128	1170	7	00:00:00.05
miles500_02.dat	128	1170	10	00:00:00.05
miles750_01.dat	128	2113	4	00:00:00.05
miles750_02.dat	128	2113	6	00:00:00.05
mug100_1_01.dat	100	166	11	00:00:00.05
mug100_1_02.dat	100	166	18	00:00:00.05
mug100_25_01.dat	100	166	15	00:00:00.05
mug100_25_02.dat	100	166	21	00:00:00.05
mug88_1_01.dat	88	146	9	00:00:00.05
mug88_1_02.dat	88	146	20	00:00:00.05
mug88_25_01.dat	88	146	7	00:00:00.05
mug88_25_02.dat	88	146	19	00:00:00.05
myciel3_01.dat	11	20	2	00:00:00.05
myciel3_02.dat	11	20	3	00:00:00.05
myciel4_01.dat	23	71	1	00:00:00.05
myciel4_02.dat	23	71	1	00:00:00.05
myciel5_01.dat	47	236	1	00:00:00.05
myciel5_02.dat	47	236	1	00:00:00.05
myciel6_01.dat	95	755	1	00:00:00.05
myciel6_02.dat	95	755	1	00:00:00.05
myciel7_01.dat	191	2360	1	00:00:00.05
myciel7_02.dat	191	2360	1	00:00:00.05

Table 9: Solutions for color benchmarks (part 3)

Dat File	n	m	Solution	Time
qg.order100_01.dat	10000	990000	97	00:00:00.26
qg.order100_02.dat	10000	990000	98	00:00:00.32
qg.order30_01.dat	900	26100	30	00:00:00.08
qg.order30_02.dat	900	26100	29	00:00:00.06
qg.order40_01.dat	1600	62400	40	00:00:00.09
qg.order40_02.dat	1600	62400	39	00:00:00.06
qg.order60_01.dat	3600	212400	59	00:00:00.09
qg.order60_02.dat	3600	212400	58	00:00:00.08
queen10_10_01.dat	100	1470	8	00:00:00.06
queen10_10_02.dat	100	1470	11	00:00:00.05
queen11_11_01.dat	121	1980	8	00:00:00.05
queen11_11_02.dat	121	1980	8	00:00:00.05
queen12_12_01.dat	144	2596	6	00:00:00.05
queen12_12_02.dat	144	2596	9	00:00:00.05
queen13_13_01.dat	169	3328	12	00:00:00.08
queen13_13_02.dat	169	3328	4	00:00:00.05
queen14_14_01.dat	196	4186	9	00:00:00.05
queen14_14_02.dat	196	4186	8	00:00:00.05
queen15_15_01.dat	225	5180	5	00:00:00.05
queen15_15_02.dat	225	5180	4	00:00:00.05
queen16_16_01.dat	256	6320	5	00:00:00.05
queen16_16_02.dat	256	6320	7	00:00:00.05
queen5_5_01.dat	25	160	7	00:00:00.05
queen5_5_02.dat	25	160	7	00:00:00.05
queen6_6_01.dat	36	290	8	00:00:00.05
queen6_6_02.dat	36	290	9	00:00:00.05
queen7_7_01.dat	49	476	11	00:00:00.05
queen7_7_02.dat	49	476	8	00:00:00.05
queen8_12_01.dat	96	1368	10	00:00:00.08
queen8_12_02.dat	96	1368	6	00:00:00.05
queen8_8_01.dat	64	728	9	00:00:00.07
queen8_8_02.dat	64	728	9	00:00:00.05
queen9_9_01.dat	81	1056	7	00:00:00.05
queen9_9_02.dat	81	1056	5	00:00:00.05
school1_01.dat	385	19095	4	00:00:00.06
school1_02.dat	385	19095	11	00:00:00.05
school1_nsh_01.dat	352	14612	4	00:00:00.07
school1_nsh_02.dat	352	14612	8	00:00:00.05
wap01a_01.dat	2368	110871	22	00:00:00.07
wap01a_02.dat	2368	110871	76	00:00:00.07
wap02a_01.dat	2464	111742	25	00:00:00.08
wap02a_02.dat	2464	111742	97	00:00:00.08
wap03a_01.dat	4730	286722	10	00:00:00.10
wap03a_02.dat	4730	286722	96	00:00:00.11
wap04a_01.dat	5231	294902	21	00:00:00.10
wap04a_02.dat	5231	294902	112	00:00:00.11
wap05a_01.dat	905	43081	16	00:00:00.07
wap05a_02.dat	905	43081	26	00:00:00.06
wap06a_01.dat	947	43571	17	00:00:00.07
wap06a_02.dat	947	43571	36	00:00:00.06
wap07a_01.dat	1809	103368	9	00:00:00.08
wap07a_02.dat	1809	103368	36	00:00:00.07
wap08a_01.dat	1870	104176	13	00:00:00.07
wap08a_02.dat	1870	104176	43	00:00:00.07
will199GPIA_01.dat	701	6772	1	00:00:00.05
will199GPIA_02.dat	701	6772	3	00:00:00.05

Table 10: Solutions for color benchmarks (part 4)

(PARIS) Planning Algorithms for Reconfiguring Independent Sets

Remo Christen[1], Salomé Eriksson[1], Michael Katz[2], Emil Keyder[3], Christian Muise[4], Alice Petrov[4], Florian Pommerening[1], Jendrik Seipp[5], Silvan Sievers[1], and David Speck[6]

[1]University of Basel
[2]IBM T.J. Watson Research Center
[3]Invitae
[4]Queen's University
[5]Linköping University
[6]University of Freiburg

1 Introduction

The general approach to all of the solver tracks was to model the ISR problem as one of automated planning, and use a selection of state-of-the-art solvers to solve these instances. Throughout this document, we describe the encoding, solvers, and overall search setup.

2 Planning Encoding

There are two main encodings we considered – *single* and *split*. The former uses a single action to move a token from one location to another, while the latter uses two actions – one to pick up a token and another to place it. While the full details on the Planning Domain Definition Language (PDDL) is out of scope for this document, the snippets presented here are fairly self-explanatory. More details on the PDDL standard can be found in [Haslum et al., 2019].

Figure 1 shows the PDDL code for a single action move, with comments interleaved. Rather than define the actions in a lifted manner, we found that generating the ground actions (along with the conditions necessary to retain the independent set) was the most effective.

Figure 2 shows the pair of actions for the split encoding. Ultimately, we found this encoding to be the most effective, and so was used for all tracks and solvers.

```
(:action move-l7-l5

    :precondition
      (and
        ; Destination free
        (free l5)

        ; Source has token
        (tokened l7)

        ; Destination's neighbours are free
        (free l3) (free l6))

    :effect (and

        ; Move the token from l7 to l5
        (not (tokened l7))
        (free l7)
        (not (free l5))
        (tokened l5))
)
```

Figure 1: PDDL example of a single action move.

```
(:action pick-l7
    :precondition
      (and
        ; Token at the location
        (tokened l7)
        ; Not holding one currently
        (handfree))
    :effect
      (and
        ; Token no longer there
        (not (tokened l7))
        (free l7)
        ; Now holding a token
        (not (handfree))
        (holding))
)
```

```
(:action place-l7
    :precondition
      (and
        ; Holding a token
        (holding)
        ; Destination is free
        (free l7)
        ; Destination neighbours are free
        (free l2))
    :effect
      (and
        ; No longer holding a token
        (not (holding))
        (handfree)
        ; Destination contains the token
        (tokened l7)
        (not (free l7)))
)
```

Figure 2: PDDL example of a pair of split actions for moving a token.

Finally, the planning systems we used are all built on a common software framework which first parses and preprocesses the PDDL into an intermediate form known as SAS+ [Bäckström and Nebel, 1995]. To save on this computational effort, we instead directly encoded the input problem instances for the ISR contest into the SAS+ format. While in general this may lead to a degradation in performance (some planning problems benefit greatly from the preprocessing), in the ISR setting it was far more effective to skip this initial phase of planner technology.

3 Engine: Core Solver or Algorithm

We use sequential algorithm portfolios for each of the three solver tracks. That is, we run a sequence of algorithms, each with an associated time limit. The next section describes the algorithms that we use in our sequential portfolios.

3.1 Our algorithms

A*+Landmarks We run an A* search [Hart et al., 1968] with a *landmark count* heuristic [Karpas and Domshlak, 2009] that uses two different kinds of landmarks: h^1 landmarks [Keyder et al., 2010] and RHW landmarks [Richter et al., 2008]. The landmark costs are combined with *uniform cost partitioning* [Katz and Domshlak, 2008].

GBFS+Landmarks We run a greedy best-first search (GBFS) [Doran and Michie, 1966] with a *landmark count* heuristic [Keyder et al., 2010] that computes all landmarks of the *delete-relaxed* task [Bonet and Geffner, 2001] (h^1 landmarks). The landmark costs are combined with *uniform cost partitioning* [Katz and Domshlak, 2008].

Symbolic search We run a forward symbolic blind search [Torralba et al., 2017, Speck et al., 2020a] using Binary Decision Diagrams [Bryant, 1986] as the underlying data structure. The symbolic planner we use is SymK [Speck et al., 2020b], which uses CUDD [Somenzi, 2015] as its decision diagram library. This algorithm is optimal, sound, and complete, i.e., if it reports a plan, this is a shortest plan, if it reports unsolvability, the task is indeed unsolvable, and given sufficient resources, it will eventually find a shortest plan.

Symbolic top-k search We run a modified forward symbolic blind search based on an algorithm called SymK-LL [von Tschammer et al., 2022], implemented in the symbolic planner SymK [Speck et al., 2020b], which iteratively finds and generates all loopless plans of a given task. However, we have made the following two major adjustments to solve the problem of finding the longest loopless plan feasible. First, once we find a goal state $s_\star$ reachable with a certain cost c, we reconstruct only one loopless plan with cost c and ignore all other plans leading to $s_\star$ or any other goal state reachable with c. Second, since the

split encoding introduced intermediate states in which a token is picked up, we ignore these artificial states when evaluating whether a plan is loopless during the plan reconstruction of SymK-LL. This algorithm has the interesting property that it iteratively finds longer plans, starting with the shortest one, and eventually finds the longest loopless plan if enough resources are available.

Numeric abstraction We abstract the problem to a numeric planning problem and check for unsolvability in the abstraction. Since this algorithm is new, we describe it in more detail in Section 4.

We now describe our sequential algorithm portfolios. Our portfolio for the *existent* track is identical to the one for the *shortest* track.

3.2 Portfolio for *shortest* and *existent* tracks

We list the algorithms and their time limits (the first to find a solution is saved and the rest of the steps ignored):

1. Numeric abstraction: 10sec
2. Symbolic search: 70min
3. A*+Landmarks: 70min
4. GBFS+Landmarks: 70min
5. Numeric abstraction: 14hr

3.3 Single best solver for *shortest* and *existent* tracks

The following single solver was used as a single-solver submission. It had the best coverage among all solvers in the portfolio.

- GBFS+Landmarks: 70min

3.4 Portfolio for *longest* track

We list the algorithms and their time limits:

1. GBFS+Landmarks: 330 seconds
2. Symbolic top-k search: 70min

As a fall-back, if neither of the above approaches produced a solution longer than the shortest/existent track, we used the solution to the shortest/existent track as a default.

3.5 Single best solver for *longest* track

The following single solver was used as a single-solver submission. It had the best coverage among all solvers in the portfolio. Note that we did not use the "fallback" option for this single track submission.

- Symbolic top-k search: 70min

4 Numeric Abstraction

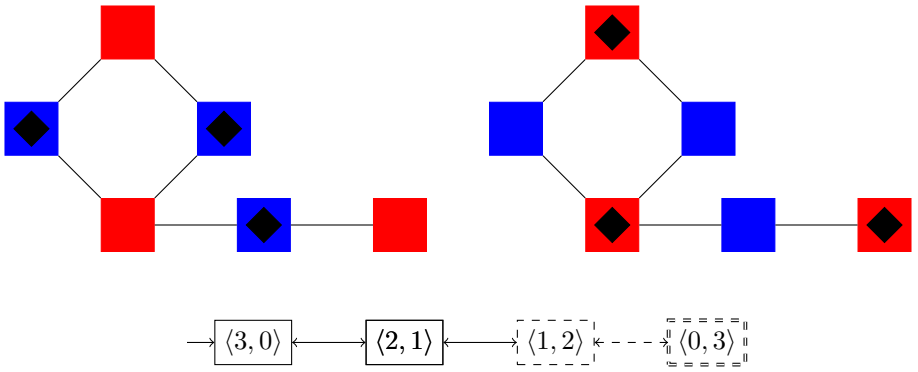


Figure 3: Example coloring for numeric abstraction. The initial state is on the left and the goal state on the right. Nodes are colored blue if they have a token in the initial state but not the goal state and red if they have no token in the initial state but a token in the goal state. The abstract state space is shown below the instance. Dashed nodes are pruned.

The *numeric abstraction* component of our solver tries to detect if the task is unsolvable by abstracting it to a numeric planning problem. Given an ISR problem, we come up with a *coloring* of the vertices in the graph, i.e., a function that maps each vertex of the graph to one color. Many different ways of coming up with a good coloring are conceivable but here we opted for a simple strategy that uses up to four colors:

- one for nodes that contain a token both in the initial state and in the goal state;
- one for nodes that contain a token only in the initial state but not in the goal state;
- one for nodes that contain a token only in the goal state but not in the initial state; and
- one for nodes that are empty in the initial state and goal state.

Colors for situations that do not occur are not used. For example, the task in Figure 3 only requires two colors with the method above.

Given a coloring, each configuration of tokens can be abstracted to a state with one numeric variable per color that counts how many token currently are on vertices with this color. For example, the initial state in Figure 3 has 3 tokens on blue nodes and 0 on red nodes, so it can be represented as the numeric state $\langle 3, 0\rangle$. The goal state has all three tokens on red nodes and none on blue, so it can represented by $\langle 0, 3\rangle$.

Moving any token from a node colored c_i to a node colored c_j changes the abstract state from $\langle c_1, \ldots, c_i, \ldots, c_j, \ldots, c_n\rangle$ to $\langle c_1, \ldots, c_i{-}1, \ldots, c_j{+}1, \ldots, c_n\rangle$. The main observation for this component is that if any solution to the full problem exists, there has to be a solution in our abstraction as well. We thus construct the state space of the abstraction in the following way.

For a state $s = \langle c_1, \ldots, c_n\rangle$, we construct one successor for each pair of different colors c_i and c_j that differs from s by a single token that moved from c_i to c_j. In our running example, the initial state $\langle 3, 0\rangle$ has a single successor $\langle 2, 1\rangle$, and this state has two successors $\langle 3, 0\rangle$ (which we skip because we have already seen this state) and $\langle 1, 2\rangle$. The latter state now has the abstract goal state $\langle 0, 3\rangle$ as a successor (see Figure 3).

Whenever we generate a state, we check whether such a state is possible (independent of whether we are actually able to reach such a state). If it is not possible to place the token on the respective colors in the way suggest, then we do not have to consider it or its successors. In our running example, the state $\langle 1, 2\rangle$ is not realizable: not matter where we place the blue token, it blocks two of the three red nodes. We use a MIP solver to check if a state s is realizable by checking if the following system of constraints have a solution:

$$
\begin{aligned}
x_i + x_j &\leq 1 \quad \text{for all edges } \langle i, j\rangle \text{ in the graph} \\
\sum_{i \in N_c} x_i &\geq s[c] \quad \text{for all colors } c \\
x_i &\in \{0, 1\} \qquad \text{for all nodes } i
\end{aligned}
$$

where N_c is a set of all nodes with color c and $s[c]$ is the amount of tokens that should be on color c in state s. The abstract state s is realizable iff the constraints have a solution.

If we generate a state that matches the goal state ($\langle 0, 3\rangle$ in our example), we know that there is an abstract path to the goal state. In this case, we still do not know if there is a real path to the goal state and return `unknown` (this component of the solver is incomplete). However, if we expand the full abstract state space without finding a path to an abstract goal state, there is no solution to the abstract problem which also means there cannot be a solution to the original problem. The abstract state space is usually small. In our running example, it only has 4 states and we only have to explore 3 of them, as we prune state $\langle 1, 2\rangle$ before reaching a goal state.

We use this component in two places: first with a small time limit at the start of the solver to handle all cases where we can quickly prove unsolvability. Then again with a large time limit after all other components to catch unsolvable instances that are hard to prove unsolvable.

5 Computation Environment

All evaluations were conducted on a single machine running Ubuntu 20.04 (evaluations done through the Docker image for the submission), with the following specs:

- **CPU**: Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz
- **Cores**: 16
- **MEM**: 128Gb

References

C. Bäckström and B. Nebel. Complexity results for SAS^+ planning. *Computational Intelligence*, 11(4):625–655, 1995.

B. Bonet and H. Geffner. Planning as heuristic search. *AIJ*, 129(1):5–33, 2001.

R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

J. E. Doran and D. Michie. Experiments with the graph traverser program. *Proceedings of the Royal Society A*, 294:235–259, 1966.

P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

P. Haslum, N. Lipovetzky, D. Magazzeni, and C. Muise. *An Introduction to the Planning Domain Definition Language*. Morgan & Claypool, 2019. ISBN 9781627058759. URL `http://www.morganclaypoolpublishers.com/catalog_Orig/product_info.php?products_id=1384`.

E. Karpas and C. Domshlak. Cost-optimal planning with landmarks. In *Proc. IJCAI 2009*, pages 1728–1733, 2009.

M. Katz and C. Domshlak. Structural patterns heuristics via fork decomposition. In *Proc. ICAPS 2008*, pages 182–189, 2008.

E. Keyder, S. Richter, and M. Helmert. Sound and complete landmarks for and/or graphs. In *Proc. ECAI 2010*, pages 335–340, 2010.

S. Richter, M. Helmert, and M. Westphal. Landmarks revisited. In *Proc. AAAI 2008*, pages 975–982, 2008.

F. Somenzi. CUDD: CU decision diagram package - release 3.0.0. https://github.com/ivmai/cudd, 2015. Accessed: 2022-03-24.

D. Speck, F. Geißer, and R. Mattmüller. When perfect is not good enough: On the search behaviour of symbolic heuristic search. In *Proc. ICAPS 2020*, pages 263–271, 2020a.

D. Speck, R. Mattmüller, and B. Nebel. Symbolic top-k planning. In *Proc. AAAI 2020*, pages 9967–9974, 2020b.

Á. Torralba, V. Alcázar, P. Kissmann, and S. Edelkamp. Efficient symbolic search for cost-optimal planning. *AIJ*, 242:52–79, 2017.

J. von Tschammer, R. Mattmüller, and D. Speck. Loopless top-k planning. In *Proc. ICAPS 2022*, 2022.

Greedy BMC Solver for the Independent Set Reconfiguration Problem

Takahisa Toda[1]

[1]Graduate School of Informatics and Engineering, The University of Electro-Communications

June 27, 2022

1 Solver Description

Our solver, named *greedy BMC solver*, is implemented as the integration of a greedy heuristic search and bounded model checking. If the greedy search finds a reconfiguration sequence in success, the solver returns it, and otherwise, goes into the BMC computation with the inititial state replaced by the end state of the "incomplete" sequence, i.e. the sequence that is not reachable to the target state. In the BMC computation, it first transforms independent-set-reconfiguration problem instances into bounded model checking instances, then applies a bounded model checker with a fixed bound, and obtains reconfiguration sequences as counterexamples to the LTL formulas [**?**]. To boost the performance, the modeling phase into BMC is done by exploiting a small edge clique cover for the input graph. The solver is often powerful in such graphs as having a large number of edges but covered by a small number of cliques, as confirmed in some benchmark instances of this challenge.

For the bounded model checking and the edge clique cover computation, our solver uses the following programs inside it.

- ECC 8: the program for the edge clique cover computation [**?**], obtained from `https://github.com/Pronte/ECC` .
- NuSMV version 2.6.0: the program for the bounded model checking [**?**], obtained from `https://nusmv.fbk.eu/`.

As is the case for bounded model checking, our solver needs a maximum number of transitions to be fixed and it is able to answer yes or no only for the existence of reconfiguration sequences of length less than or equal to the step size. As stated above, our solver is combined with the heuristic search and hence, the obtained sequence are no longer quaranteed to be the shortest.

2 Metric

- (solver#existent)
- (solver#shortest)
- (solver#longest)

3 Computation Environment

- Intel(R) Xeon(R) CPU E5-2609 v4 @ 1.70GHz
- 32G bytes memory
- Ubuntu 20.04

4 Usage

ECC8.jar and NuSMV are not included in this submission:

```
~/2022solver# ls bin/
col2nde  compact  ecc2mat  greedy  greedy_bmc-solver_ecc.sh
id-tj-ecc  out2ans_uniform.awk  postproc  st2ltl.awk
```

So, at first, obtain the above two programs and place them in bin.

Check if there exists tmp in /2022solver, where our solver generates various temporary files.

```
~/2022solver$ ls
README.md  bin  example  run.sh  tmp
```

Run run.sh as follows.

```
~/2022solver$ ./run.sh example/hc-toyyes-01.col example/hc-toyyes-01_01.dat
```

ISR Solver Track Documentation

Ulrick BLÉ[1], Xiaoteng CUI[1], Feibiao WU[1], and Zijie ZHONG[1]

[1]École Centrale de Nantes

1 Engine: Core Solver / Algorithm

We employed A algorithm for small scale test cases in order to obtain the results with brute force. With test cases of larger scale, SARSA algorithm is used instead.

recongo: ASP-based Combinatorial Reconfiguration Problem Solver

Yuya Yamada[1], Masato Kato[2], Shuji Kosuge[2], Raito Takeuchi[1], and Mutsunori Banbara[1]

[1]Graduate School of Informatics, Nagoya University, Japan
[2]School of Informatics, Nagoya University, Japan

1 Metric

- solver#existent

2 Engine: Core Solver or Algorithm

Answer Set Programming (ASP) is an approach to declarative problem solving. ASP provides a rich language and is well suited for modeling combinatorial problems in Artificial Intelligence and Computer Science. Recent improvements in the effectiveness of ASP systems have encouraged researchers to use ASP for solving problems in diverse areas, such as automated planning, constraint satisfaction, model checking, etc.

The *recongo* solver is an ASP-based combinatorial reconfiguration problem solver. *recongo* reads a problem instance and converts it into ASP facts. In turn, these facts are combined with a first-order encoding for problem solving, which is subsequently solved by a general-purpose ASP system, in our case *clingo*. For CoRe Challenge 2022, we develop a basic ASP encoding and some additional hints for solving independent set reconfiguration problems.

- single-engine

3 Computation Environment

- OS: Mac OS
- CPU: 3.3GHz 12core Intel Xeon W
- MEM: 96GB
- ASP system: *clingo*-5.5.1 [1]

[1] `https://potassco.org/clingo/`

ReconfAIGERation entering Core Challenge 2022

Nils Froleyks[1], Emily Yu[1], and Armin Biere[2]

[1]Johannes Kepler University, Linz, Austria
[2]Albert Ludwig University, Freiburg, Germany

March 31, 2022

Abstract

ReconfAIGERation encodes the Independent Set Reconfiguration problem into a circuit in AIGER format. The bad state property encodes that the target independent set is reached, without violating the token jump rule. A witness for the model checking problem is decoded into a reconfiguration sequence.

Encoding and Solving

To encode the ISR problem into AIGER (version 1.9 [1]) we define a circuit with one latch and two inputs per node. The latches encode if the node is in the current independent set. The inputs are divided into two groups: freed and marked. At each step, exactly one[1] node in the current IS is freed and one not in the current IS marked, thus enforcing the token jump rule. The independent set constraint is easily encoded by combining gates for each edge in the graph.

The tool aigtoaig is used to translate the human readable output from the previous step (.aag) into a binary encoded format (.aig).

For the existent-track the Aiger enconding is solved by the model checker ABC [4]. ABC runs multiple model checking algorithms in parallel. Among them is an implementation of IC3/PDR [3]. It can therefore prove the unsolvability of a problem.

For the shortest-track the model checking problem is solved with the Bounded Model Checker CaMiCaL [6]. In Bounded Model Checking (BMC)

[1]To encode the at-most-one part we used the product encoding [5] since a naive square encoding runs out of memory before the encoding is completed on some of the instances.

the transition function of a circuit is encoded into a SAT formula and copied a number of times (called makespan). The formula is satisfiable exactly if the target IS is reachable in at most makespan-many steps. Since the makespan is incremented step-wise, the first solution found is guaranteed to be the shortest. CaMiCaL is highly integrated with the incremental SAT solver CaDiCaL [2] to optimize the effectiveness of incremental inprocessing [6].

For the longest-track we skip the encoding to Aiger and use CaDiCaL directly. In addition to the usual BMC encoding we add a uniqueness constraint to the steps of the solution to disallow loops in the reconfiguration sequence. In theory the makespan is increased until it reaches the maximum possible unique solution length: $N = \binom{\text{number of nodes}}{\text{size IS}}$. The last found solution is then guaranteed to be the longest. In practice N exceeds the maximum integer value in most cases and the computation will not terminate in reasonable time or memory.

Implementation

Our solver is written in Nim and can be compiled with:

```
nim cpp -d:release --passL:cadical/build/libcadical.a reconfaigeration
```

It depends on other executables in the same directory. We cannot provide the source code for all of them.

We generated the log files in parallel on 32 nodes of our cluster. Each node has access to two 8-core Intel Xeon E5-2620 v4 CPUs running at 2.10 GHz (turbo-mode disabled) and 128 GB main memory.

Since ABC spawns multiple processes we only allocated one instance per node. For CaMiCaL we ran 4 instances in parallel on each node. The memory was limited to 32 GB for each of the instances. The timeout was set to 10000 seconds for both. For the longest-track we used the same number of instances and memory limit and a timeout of 5000 seconds.

References

[1] Armin Biere, Keijo Heljanko, and Siert Wieringa. *AIGER 1.9 and Beyond*. 11/2. Altenbergerstr. 69, 4040 Linz, Austria: Institute for Formal Models and Verification, Johannes Kepler University, July 2011, 2011.

[2] Armin Biere et al. "Entering the SAT Competition 2020". In: (2020), p. 4.

[3] Aaron R. Bradley. "SAT-Based Model Checking without Unrolling". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Ranjit Jhala and David Schmidt. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 70–87. ISBN: 978-3-642-18275-4. DOI: 10.1007/978-3-642-18275-4_7.

[4] Robert Brayton and Alan Mishchenko. "ABC: An Academic Industrial-Strength Verification Tool". In: *Computer Aided Verification*. Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 24–40. ISBN: 978-3-642-14295-6. DOI: 10.1007/978-3-642-14295-6_5.

[5] Jingchao Chen. "A New SAT Encoding of the At-Most-One Constraint". In: *Proc. Constraint Modelling and Reformulation* (2010), p. 8.

[6] Katalin Fazekas, Armin Biere, and Christoph Scholl. "Incremental Inprocessing in SAT Solving". In: *Theory and Applications of Satisfiability Testing – SAT 2019*. Ed. by Mikoláš Janota and Inês Lynce. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 136–154. ISBN: 978-3-030-24258-9. DOI: 10.1007/978-3-030-24258-9_9.

A decision diagram-based solver for the independent set reconfiguration problem

Jun Kawahara[1] and Hiroki Yamazaki[1]

[1]Kyoto University

1 Engine: Core Solver or Algorithm

Our solver is based on zero-suppressed binary decision diagrams [**?**] and the graphillion library [**?**].

References

[1] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *the 30th ACM/IEEE design automation conference*, pages 272–277, 1993. https://doi.org/10.1145/157485.164890

[2] T. Inoue, H. Iwashita, J. Kawahara, S. Minato, Graphillion: software library for very large sets of labeled graphs, International Journal on Software Tools for Technology Transfer 18 (1) 57–66, 2016. https://doi.org/10.1007/s10009-014-0352-z

Graph track description

Nicolas Bousquet[1], Bastien Durain[2], and Théo Pierron[1]

[1]Univ. Lyon, Université Lyon 1, LIRIS UMR CNRS 5205,
F-69621, Lyon, France
[2]ENS Lyon

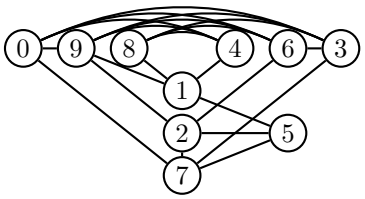
1 Results

The sizes of the reconfiguration sequences in each track are given in the table below.

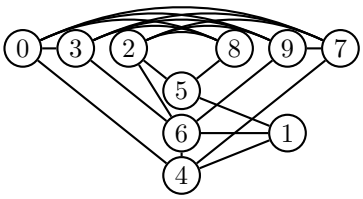
Size n of the graph	Size k of the IS	Length of the sequence
10	4	10
50	15	3410
100	30	3495250

2 Construction for $n = 10$

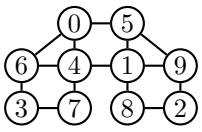
By bruteforcing all graphs on $n = 10$ vertices, we obtain that three graphs maximize the diameter of the reconfiguration graph. In each case, the reconfiguration graph is a path on 11 vertices. The three graphs are depicted below:



(a) Graph G_1, maximum distance obtained between 167 and 789.



(b) Graph G_2, maximum distance obtained between 012 and 056.



(c) Graph G_3, maximum distance obtained between 0123 and 2345.

3 Construction for $n = 50$ and $n = 100$

The construction for higher number of vertices is based on the graph G_1 depicted before. We draw below the reconfiguration graph of 3-IS in G_1.

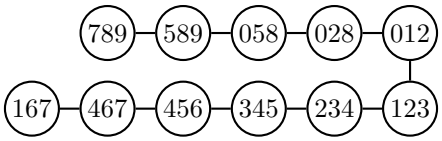


Figure 2: The reconfiguration graph of 3-IS in G_1.

Our construction relies on the following operation. Starting from an instance (G, α, β), we construct an instance (G', α', β') as follows:

- Add 10 new vertices to G inducing G_1. We assume that these vertices are labeled with $0, \ldots, 9$ according to Figure 1a.
- For every vertex u of G not in α, we add an edge in G' between u and the vertex 1 of G_1. We also add, for every vertex $v \in G$ not in β an edge in G' between v and 5.
- We set $\alpha' = \alpha \cup \{7, 8, 9\}$ and $\beta' = \alpha \cup \{7, 8, 9\}$.

We claim that this construction satisfies the following.

- $|V(G')| = |V(G)| + 10$;
- $|\alpha'| = |\alpha| + 3$ and α' is a maximum independent set in G';
- The connected component of the reconfiguration graph of G containing α' and β' is a path whose endpoints are α' and β'.
- $d' = 4d + 10$ where d (resp. d') is the distance between α and β (resp. α' and β') in the reconfiguration graph of G (resp. G').

The first point is straightforward. The second point also holds. Indeed, α' is a maximum independent set since α is a maximum independent set of G and G_1 has independence number 3. Since α is a maximum independent set of G, there must always be three tokens in G_1, and these tokens can only move following the reconfiguration sequence of G_1. Therefore, at each step, one can either move a token in G (following the reconfiguration graph of G) or in G_1.

Now observe that tokens in G_1 cannot move unless the tokens in G induce α or β. Conversely, tokens in G cannot move unless the tokens in G_1 induce an IS not containing 1 nor 5. This ensures that the reconfiguration graph of G' is a path with endpoints α' and β'.

One can finally check that the length d' of this path is $4d+10$. In order to put tokens in position 028 in G_1, one needs to put a token in vertex 5 (see Figure 2), which implies to put all tokens of G in β. This requires d steps, hence in $d + 3$

steps, we manage to obtain the independent set $\beta \cup \{0, 2, 8\}$. Now, similarly, to put the tokens on 234, we need to put tokens in 1, which requires to put back the tokens of G on α. By iterating again this argument twice, we finally obtain a transformation where the total number of token slides in G is $4d$ times, while the total number of token moves in G_1 is 10, which completes the proof.

In particular, applying several times this construction starting from the instance $(G_1, \{7, 8, 9\}, \{1, 6, 7\})$, we can construct instances (G, α, β) where G has n vertices, α, β are independent sets of size $3n/10$ at distance $\frac{10}{3}(4^{n/10} - 1)$ in the reconfiguration graph (note that their component is a path).

Note that the same construction works when replacing G_1 by the complement of a path on 5 vertices, linking the middle vertex to β. However, in that case we get $|V(G')| = |V(G)| + 5$ and $d' = 2d + 3$, hence we get graphs on n vertices with a reconfiguration sequence of size $3 \times (2^{n/5} - 1)$, which is slightly worse. This also means that this construction can be improved with a better choice of G_1.

Every Reconfiguration Starts with a First Step

Akira Suzuki[1]

[1]Graduate School of Information Sciences, Tohoku University

1 Our instance for (graph#10) track

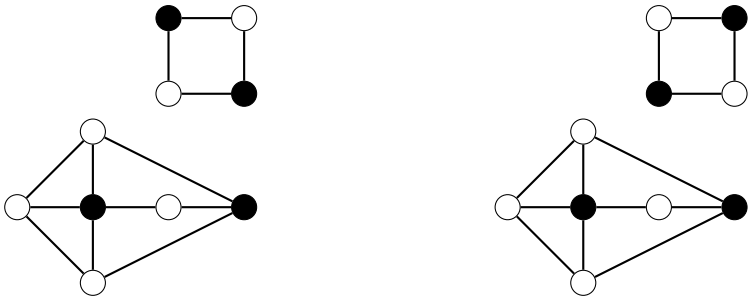
- A completely handcrafted instance, as shown in Figure 1. Our graph has two parts: the C_4 part above and the "kite" part below.
- The only difference between the initial and the target solutions is the two tokens on the C_4. However, these tokens interfere with each other, so they cannot be changed directly on the C_4 alone. Therefore, first, we need to move one of these tokens onto the kite, using four steps. (See Figure 2.)
- After reconfiguring the tokens on the C_4, we need to replay the previous step in reverse to restore the tokens on the kite, so the reconfiguration takes a total of nine steps.

2 Our instance for (graph#50) track

- Our instance is shown in Figure 3. It contains 50 vertices and 263 edges.
- We use the same idea as in our previous paper: Akira Suzuki, Amer E. Mouawad and Naomi Nishimura, Reconfiguration of dominating sets, Journal of Combinatorial Optimization 32(4), pp. 1182–1195, 2016.
- The only difference between the initial and the target solutions is the top right two tokens of Figure 3.
- However, 643 steps are required before and after reconfiguring these two tokens. Then the total reconfiguration takes 1,289 steps.

3 Our instance for (graph#100) track

- Our instance is shown in Figure 4. It contains 100 vertices and 637 edges.
- The idea is the same as (graph#50): While the only difference between the initial and the target solutions is the top left two tokens of Figure 4, the total reconfiguration takes 288,677 steps.



(a) (b)

Figure 1: Our instance for (graph#10) track, (a) the initial independent set, and (b) the target independent set.

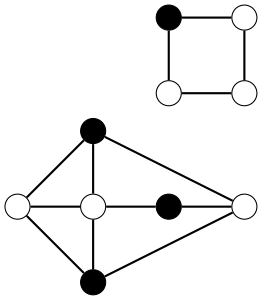
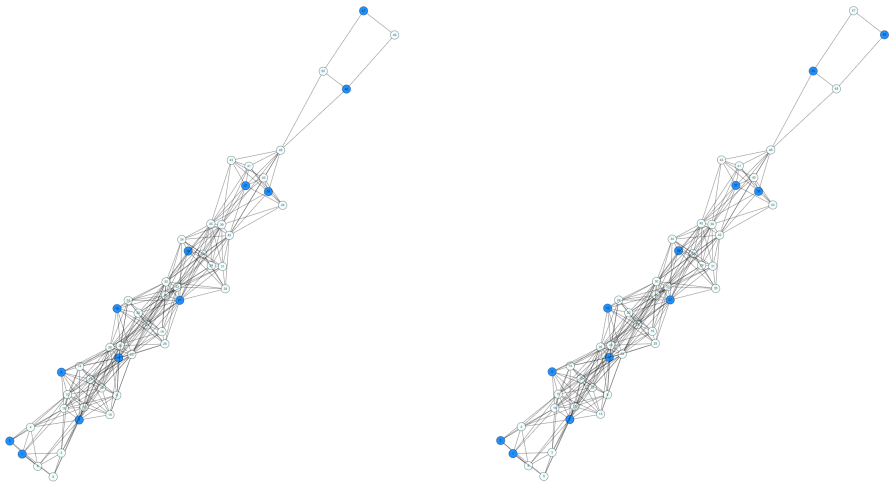
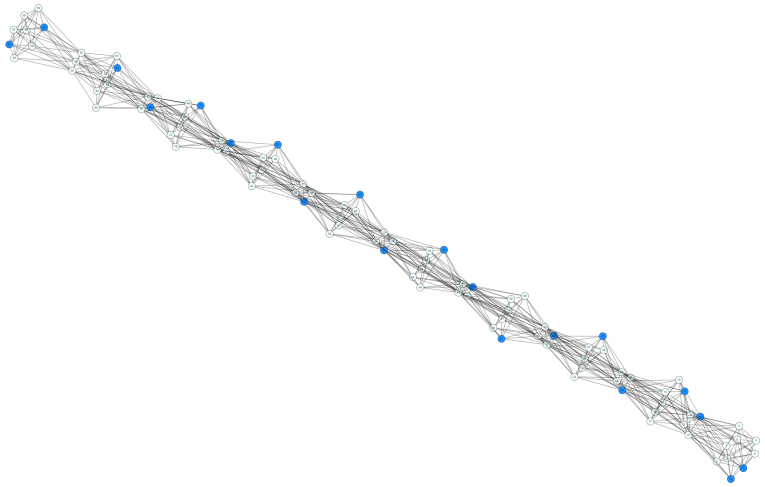


Figure 2: The independent set of four steps after the initial solution. Since there are three tokens on the kite, we can move the token on the C_4.

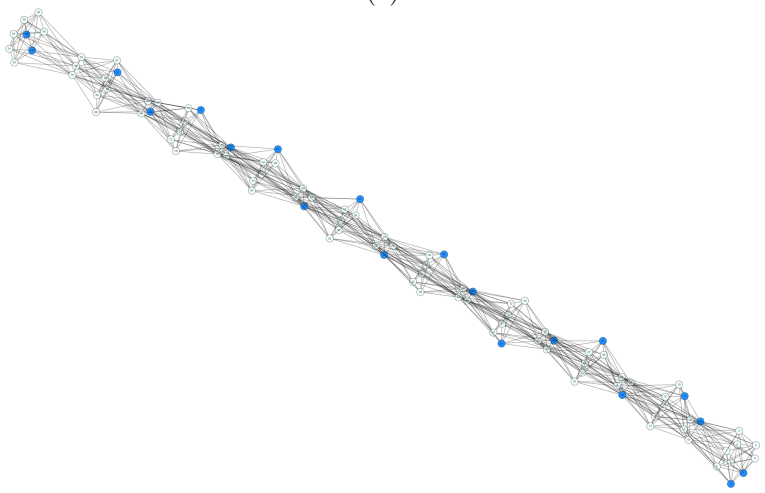


(a) (b)

Figure 3: Our instance for (graph#50) track, (a) the initial independent set, and (b) the target independent set.



(a)



(b)

Figure 4: Our instance for (graph#100) track, (a) the initial independent set, and (b) the target independent set.

A Series of Graphs With Exponentially Growing Reconfigurations Sequences of Independent Sets

Volker Turau and Christoph Weyer

Institute of Telematics, Hamburg University of Technology
Hamburg, Germany
turau@tuhh.de

1 Introduction

In this note we construct a sequence of graphs G_c with ($c \in \mathbb{N}$) together with two independent sets S_c and T_c such that the shortest reconfiguration sequence between S_c and T_c grows exponentially with respect to the size of the graphs. Reconfiguration of independent sets is with respect to the token jumping rule, i.e., a token can *jump* from one node to any other node as long as the independent set property is retained. In particular the graph G_c has size $10c$, the independent sets have size $4c$, and the shortest reconfiguration sequence for G_c has length $5(3^c - 1)$. Table 1 shows the length of the sequences for $c = 1, 5, 10$.

Size of graph	Length of reconfiguration sequence
10	10
50	1210
100	295240

Table 1: Length of reconfiguration sequences for the graphs G_c.

2 The Graph Series G_c

The graphs of the sequence we construct are called G_c with $c \in \mathbb{N}$. The basis is the graph G_1, i.e., $c = 1$. This graph is shown in Fig. 1. The independent set consists of four nodes depicted in blue, $S_c = \{2, 4, 7, 9\}$ (resp. $T_c = \{2, 5, 7, 10\}$) is on the left (resp. right). Note that these are maximum independent sets of G_1.

Table 2 shows the moves of the shortest reconfiguration sequence from S_1 to T_1 in G_1, is has length 10. Note that $S_1 \cap T_1 = \{2, 7\}$, i.e., only the tokens of

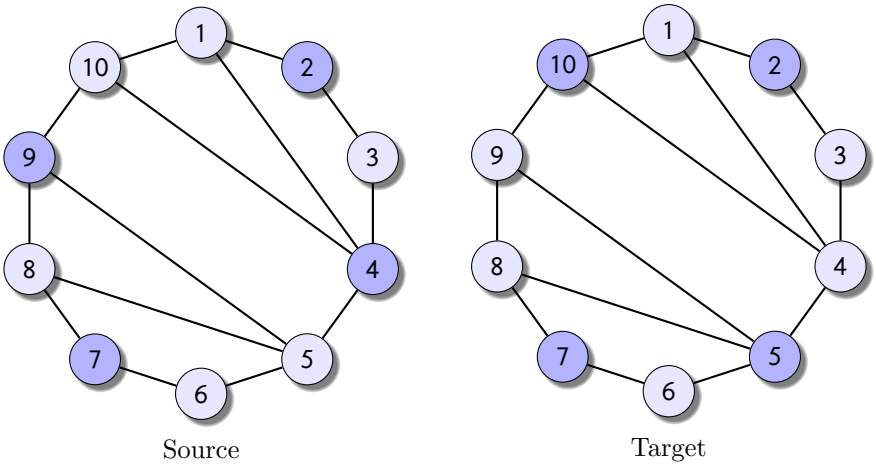


Figure 1: The base graph G_1 with 10 nodes. The blue nodes depict the start resp. the target independent set.

nodes 4 and 9 have to be move to 5 and 10. None of these moves can be done in the initial configuration. The first two moves make moving the token from 4 to 10 possible. These moves are rolled back in moves 6 and 7. Moves 4 and 5 paved the way to move the token from 9 to 5. These movements are undone in the last two moves. Note that the four *chords* block shorter reconfiguration sequences. Removing either chord $(4, 1)$ or $(5, 8)$ would allow a reconfiguration sequence of length 5 and removing any of the other two reconfiguration chords would even allow a length 2 sequence. Thus, the chords stretch the shortest reconfiguration sequences. Adding a fifth chord makes a reconfiguration impossible. The outstanding property of G_1 with respect to S_1 is that there is always only one possibility for a jump, i.e., the corresponding reconfiguration graph is a path.

These observations are the underlying idea of constructing the sequences of graphs G_c. The constructing process consists of two steps we called *duplication* and *repetition* process.

3 The Duplication Process

The duplication process starts by taking two copies of G_1. Let $\hat{G_1} = G_1 \cup \overline{G_1}$, where $\overline{G_1}$ is isomorphic to G_1. The overline operator means that the labels of the nodes are incremented by 10, i.e., the nodes of $\overline{G_1}$ are labeled $11, 12, \ldots 20$. Note that $\hat{G_1}$ is not connected. We also construct two independent sets of $\hat{G_1}$: $\hat{S_1} = S_1 \cup \overline{S_1}$ and $\hat{T_1} = T_1 \cup \overline{T_1}$.

Since S_1 resp. $\overline{S_1}$ are maximum independent sets of G_1 resp. $\overline{G_1}$ it is impossible to move a token from G_1 to $\overline{G_1}$ or vice versa. Obviously the length of the shortest reconfiguration sequence from $\hat{S_1}$ to $\hat{T_1}$ in $\hat{G_1}$ is twice as long as

#	Independent set	Jump
	2 4 7 9	
1	2 4 6 9	$7 \to 6$
2	2 4 6 8	$9 \to 8$
3	2 6 8 10	$4 \to 10$
4	3 6 8 10	$2 \to 3$
5	1 3 6 8	$10 \to 1$
6	1 3 6 9	$8 \to 9$
7	1 3 7 9	$6 \to 7$
8	1 3 5 7	$9 \to 5$
9	3 5 7 10	$1 \to 10$
10	2 5 7 10	$3 \to 2$

Table 2: A shortest reconfiguration sequence from S_1 to T_1 in G_1 has length 10.

that from S_1 to T_1 in G_1. We call this reconfiguration sequence the *canonical* sequence.

In order stretch the reconfiguration sequence from $\hat{S}_1$ to $\hat{T}_1$ we insert a kind of chords into $\hat{G}_1$, these are edges from a node in G_1 to a node in $\overline{G_1}$. The intention of inserting these chords is to block the moves of the canonical sequence.

As stated above a token from G_1 (resp. $\overline{G_1}$) can only jump to a node in G_1 (resp. $\overline{G_1}$). Hence, a reconfiguration sequence of $\hat{G}_1$ restricted to the nodes of G_1 (resp. $\overline{G_1}$) yields a valid reconfiguration sequence of G_1 (resp. $\overline{G_1}$). This property remains true even if we insert chords into $\hat{G}_1$.

The chords are inserted in such a way such that the shortest reconfiguration sequence $\mathcal{S}$ of the resulting graph when restricted to G_1 is equal to original shortest reconfiguration sequence of the original graph G_1, i.e., $\mathcal{S}|_{G_1}$ is equal to the reconfiguration sequence that is depicted in Table 2. There are eight chords inserted: $(9, 11), (9, 13), (9, 15), (9, 16), (10, 11), (10, 13), (10, 18)$, and $(10, 19)$. Denote this set of edges by C. The resulting graph is the graph G_2, it consists of 20 nodes and 36 edges (see Fig. 2). The first 26 moves of the reconfiguration sequence from $\hat{S}_1$ to $\hat{T}_1$ in G_2 are shown in Tab. 3.

Of course some moves of $\mathcal{S}$ do not change a token of G_1. So after removing duplicates $\mathcal{S}|_{G_1}$ consists of the 10 moves shown in Table 2. On the other hand $\mathcal{S}|_{\overline{G_1}}$ consists of 30 moves. The first 10 moves also correspond to the moves of Table 2 (these 10 moves are highlighted in Tab. 2). The next 10 moves are also equal to these moves but in inverse order (also highlighted in Tab. 2). Finally, the last 10 moves again correspond to the moves of Table 2. Thus, all together we have 40 moves for G_2.

4 The Repetition Process

The graphs G_c for $c > 2$ are defined inductively. G_{c+1} consists of a copy of G_c and a copy of G_1. The nodes of the copy of G_1 are labeled from $10c{+}1$ to $10c{+}10$.

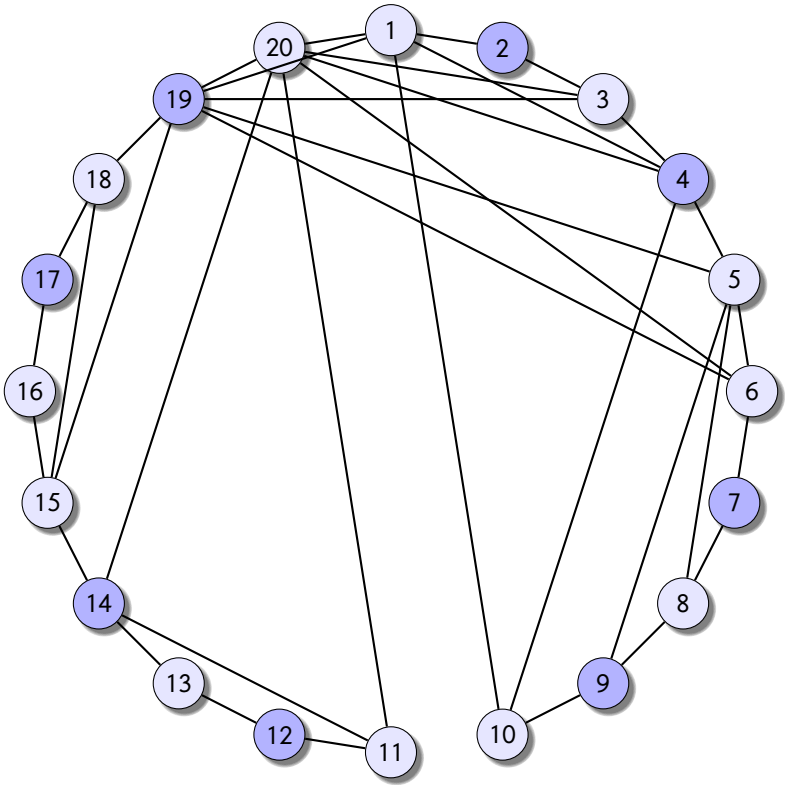


Figure 2: The graph G_2 with 20 nodes.

In addition G_{c+1} contains for each edge $(a, b) \in C$ an edge $(a+10(c-1), b+10c)$. Similarly, we extend the start and target independent set of G_c by a transformed copy (i.e., labels incremented by $10c$) of the nodes of the corresponding sets of G_1 to independent sets of G_{c+1}.

Let $\mathcal{S}$ be a shortest reconfiguration sequence of G_{c+1}. Then $\mathcal{S}$ restricted to each of the copies of G_1 in G_{c+1} is a reconfiguration sequence from S_1 to T_1. As shown above, the sequence oscillates between S_1 and T_1. Each simple such sequence in the i^{th} copy corresponds to three simple sequences in the $(i+1)^{th}$ copy. Thus, the number of moves of G_c is

$$10 \sum_{i=0}^{c-1} 3^i = 5(3^c - 1).$$

5 Discussion

The graph G_1 is constructed from a graph C_5 which is a cycle with five nodes and a single chord. This graph C_5 is smallest graph with a non-trivial reconfiguration

#	Independent set	Jump	#	Independent set	Jump
	2 4 7 9 12 14 17 19				
1	2 4 6 9 12 14 17 19	7→ 6	14	3 6 8 10 12 15 17 20	2 → 3
2	2 4 6 8 12 14 17 19	9 → 8	15	1 3 6 8 12 15 17 20	10 → 1
3	2 4 6 8 12 14 16 19	17 → 16	16	1 3 6 8 13 15 17 20	12 → 13
4	2 4 6 8 12 14 16 18	19 → 18	17	1 3 6 8 11 13 15 17	20 → 11
5	2 4 6 8 12 16 18 20	14 → 20	18	1 3 6 8 11 13 17 19	15→ 19
6	2 4 6 8 13 16 18 20	12 → 13	19	1 3 6 8 11 13 16 19	17 → 16
7	2 4 6 8 11 13 16 18	20 → 11	20	1 3 6 8 11 13 16 18	19 → 18
8	2 4 6 8 11 13 16 19	18 → 19	21	1 3 6 8 13 16 18 20	11 → 20
9	2 4 6 8 11 13 17 19	16 → 17	22	1 3 6 8 12 16 18 20	13 → 12
10	2 4 6 8 11 13 15 17	19 → 15	23	1 3 6 8 12 14 16 18	20 → 14
11	2 4 6 8 13 15 17 20	11 → 20	24	1 3 6 8 12 14 16 19	18 → 19
12	2 4 6 8 12 15 17 20	13 → 12	25	1 3 6 8 12 14 17 19	16 → 17
13	2 6 8 10 12 15 17 20	4 → 10	26	1 3 6 9 12 14 17 19	8 → 9

Table 3: A shortest reconfiguration sequence from $\hat{S}_1$ to $\hat{T}_1$ in G_2 has length 40.

sequence. The construction is analog to the described duplication process with one exception. In the copy of C_5 the start and target independent sets are interchanged.

There are a few open questions. Can the described techniques of duplication and repetition used to construct graphs with even longer reconfiguration sequences? For example with reconfiguration sequences of length $d^{O(n)}$ with $d > 3$ or even d arbitrarily large? Finally, what are better techniques to construct *good* graphs?

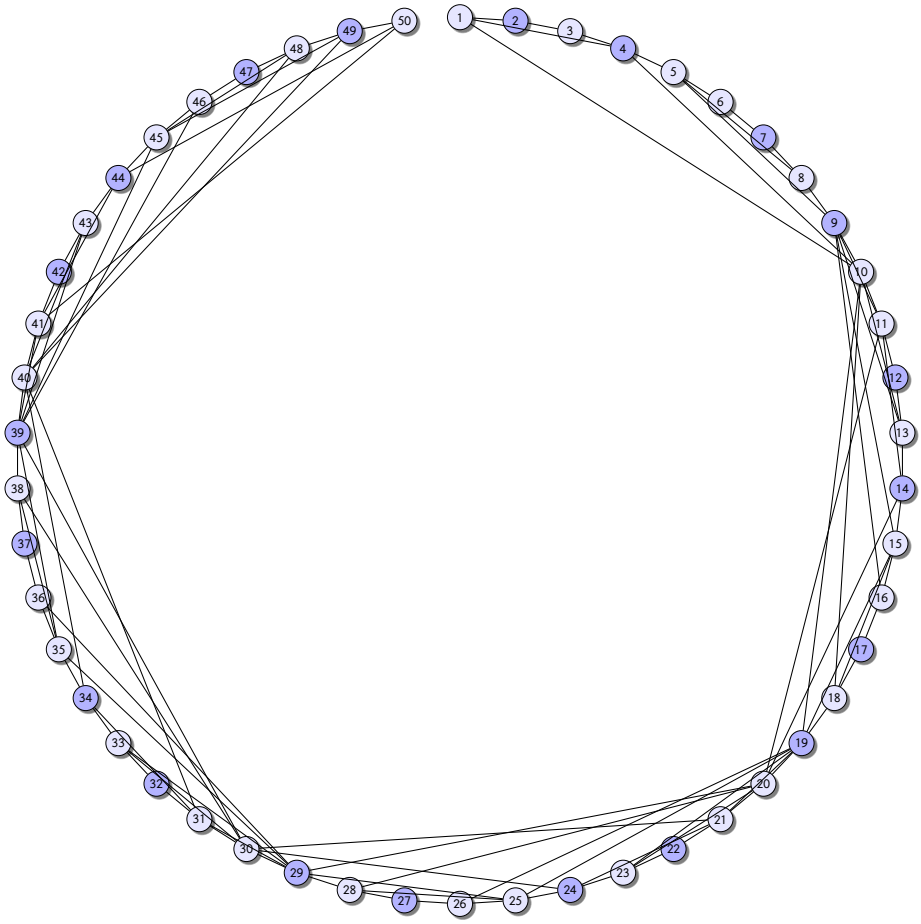


Figure 3: The graph G_5 with 50 nodes.

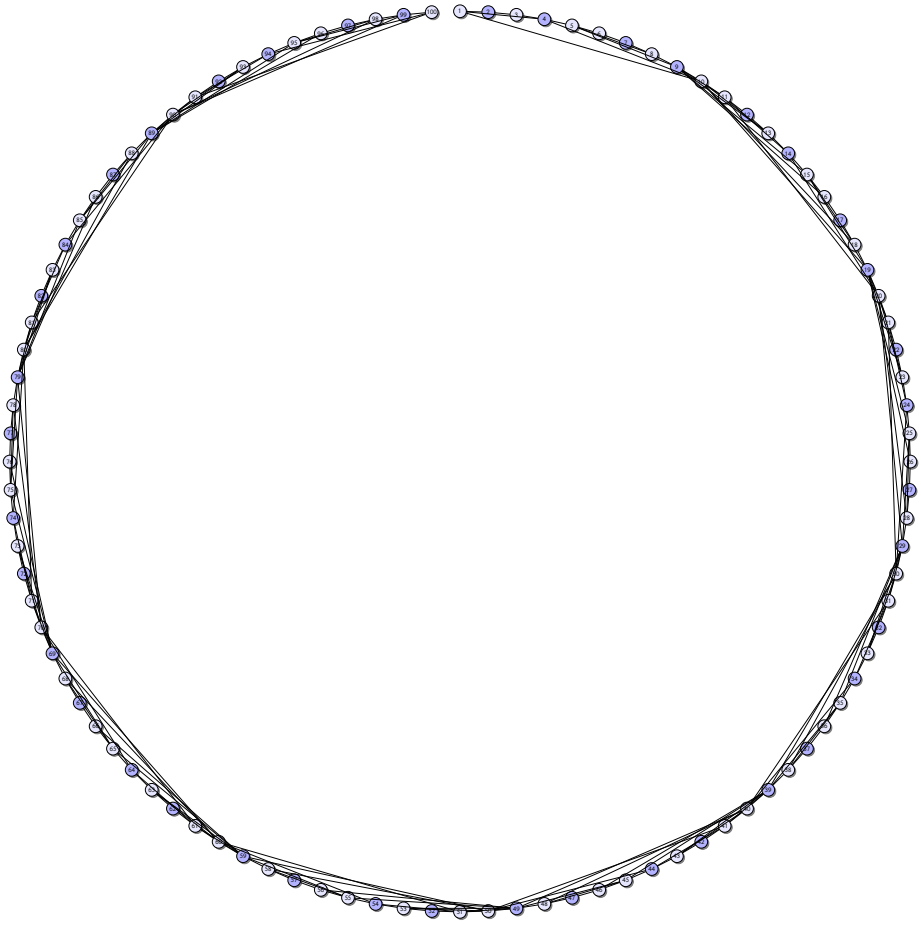


Figure 4: The graph G_{10} with 100 nodes.

Every House on the Block: A generalized solution to creating ISR instances of large plan length

Remo Christen[1], Salomé Eriksson[1], Michael Katz[2], Emil Keyder[3], Christian Muise[4], Alice Petrov[4], Florian Pommerening[1], Jendrik Seipp[5], Silvan Sievers[1], and David Speck[6]

[1]University of Basel
[2]IBM T.J. Watson Research Center
[3]Invitae
[4]Queen's University
[5]Linköping University
[6]University of Freiburg

1 Introduction

Several strategies were tried throughout the course of the contest. From hand-crafted examples to exhaustive enumeration of smaller graphs with the aid of SAT solvers and knowledge compilers. Ultimately, a useful pattern for repeated movements was discovered. Since it is readily scalable, this is what is used for the submission. Throughout the remainder of the document, we describe the core concepts that make up the graph.

2 The "House" Widget

In order to encode bit flips in a graph, we leverage a five node subgraph we call the "house widget": a 4-cycle with two adjacent nodes leading to a 5th (essentially, a triangle sitting on top of a square). The house widget has a number of properties that make it ideal to use as a building block in creating exponential sequences.

1. The graph has an optimal "long" shortest reconfiguration sequence for ISR instances of order 5.
2. You can only place two tokens on this widget, meaning each step of the reconfiguration sequence consists of a maximum independent set. In other words, the sequence is "tight" and no additional nodes can be added to the independent set at any point.

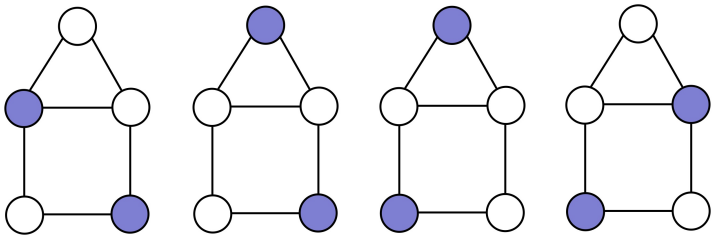


Figure 1: Reconfiguration sequence from "off" to "on"

3. The topmost node, which we call the "anchor", is occupied throughout the entire sequence with the exception of the starting state and ending state, and is required to switch the corners that the two tokens are on.

4. The sequence is unique. Thus, the solution space is a path and the behaviour of the widget is predictable.

In summary, we have a widget that remembers its setting, takes five nodes to do so, and three moves to make it happen. We call one setting "on" and the other setting "off" (they're symmetric).

3 Connecting Anchors

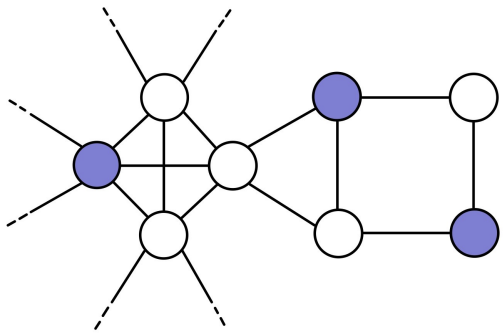


Figure 2: Four connected anchors; clearly, only one house can "flip" at a time

Since we treat our house widgets as bit flips, the first step in establishing an optimally long sequence is ensuring at most one house can be switching states at any given time. This is done by fully connecting the anchors of all houses in our graph. Recall that the anchor is both required to switch a house from "on"

to "off" and is occupied throughout the sequence. By making the anchors a fully connected subgraph, we guarantee no houses switch states simultaneously.

4 Flipping Bits

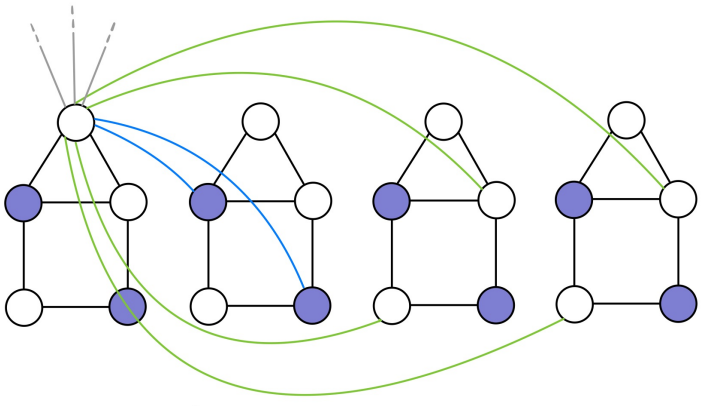


Figure 3: Edges for a single house. Blue and green edges correspond to Rules 1 and 2 respectively. Grey edges correspond to connections with other anchors.

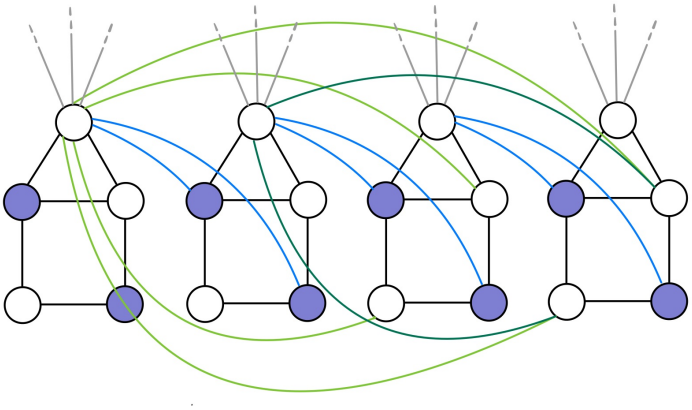


Figure 4: Four connected houses. Blue and green edges correspond to Rules 1 and 2 respectively. Assume all anchors are fully connected.

Now that we have a rigid set of widgets (referred to henceforth as bits) which we can turn "on" and "off", we must connect them in a manner that results in exponential growth. Assume bits are ordered (so there is a 1st bit, 2nd bit,

etc..). The key challenge is that any edge between house bases will permanently rule out a pair of bit configurations, thus all additional inter-house edges are between an anchor and bit values of another house. The order of bit flips is then enforced by the following relations in order to flip bit k.

Rule 1: Bit $k+1$ must be "on".

Rule 2: Bits $k+2 \cdots n$ must all be "off".

This sequence results in an exponential cost flipping things back and forth in order to get a low bit flipped.

5 Generating Graphs

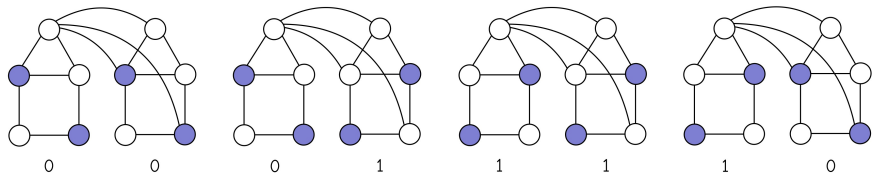


Figure 5: A series of bit flips for a graph of order 10. Note that each bit flip takes 3 moves, so the entire sequence is of length 9.

Putting everything together, graphs are generated as follows:

1. Create k "houses"
2. Make the anchors of each house a fully connected subgraph of order k
3. Add edges corresponding to the three bit flipping rules listed in Section 4, *Flipping Bits*

6 Adding Widgets

The final step is the strategy for adding an extra house. Suppose we have a sequence of k houses generated via the strategy above. We add house k+1 according to the following:

1. We can only flip house k+1 when the goal of house k is satisfied.
2. The new goal is the initial state of the k house sequence, combined with the flip of house k+1

This forces the plan length to double with each new house: achieve the old goal of the k house sequence, flip house k+1, go back to the initial state of the k house sequence.

7 Contest Instances

Since each size was a multiple of 5, the submissions for the graph track take 2, 10, and 20 houses respectively for graphs of order n=10, 50, and 100.

The lengths of the optimal plans found are as follows:

n	Length
10	9
50	3,069
100	3,145,725

The Documentation of Graph Track in Core Challenge

Takashima Yuya[1] and Yamaoka Chuta[1]

[1]Minato laboratory of Kyoto University

31 March 2022

1 Graph Structure

Consider a piece consisting of 5 vertices and 2 tokens (Fig. 1). This piece can switch between 0-state and 1-state via transition-state as shown in Figure 2.
Here, if vertex 1 is blocked by an external token in the 0-state or 1-state state, this piece will not be able to switch states (Fig. 3). By preparing piece-1 and piece-2 and connecting them with edges as shown in Figure 4, it is possible to make the possibility of transition of piece-2 depend on piece-1.
Prepare n pieces piece-1, piece-2 ... piece-n. For $i(2 \leq i \leq n)$, stretch the edges so that piece-i is transitive only when piece-$i-1$ is 1-state and piece-$j(1 \leq j \leq i-2)$ is 0-state. The initial token placement state is such that each piece is 0-state. The target token placement state is set so that only piece-n is 1-state and all other pieces are 0-state.
Suppose there are m pieces, and the state of each piece is represented by an 01-string of length m. In the transition from the initial state to the target state, this 01-string changes by one character, and all 2^m possible 01-strings are transitioned as in the Gray code.

2 Reconfiguration Sequence Length

Each piece requires 3 transitions to change its state. There are $|V|/5$ pieces in the graph, therefore the reconfiguration sequence length is $3 \times (2^{|V|/5} - 1)$.

10 vertices: 2 pieces, so the length is $3 \times (2^2 - 1) = 9$.
50 vertices: 10 pieces, so the length is $3 \times (2^{10} - 1) = 3069$.
100 vertices: 20 pieces, so the length is $3 \times (2^{20} - 1) = 3145725$.

Fig.1 A piece consisting of 5 vertices and 2 tokens

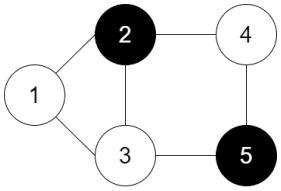


Fig.2 Transition of a piece

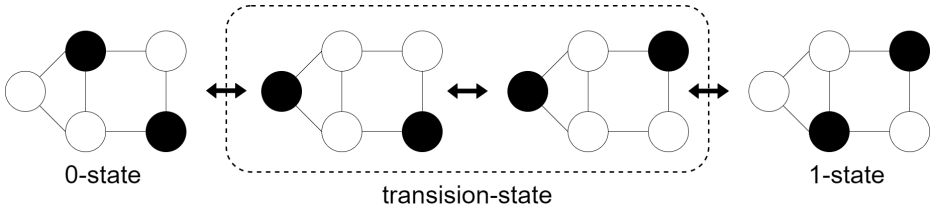


Fig.3 Blocking transitions by external tokens

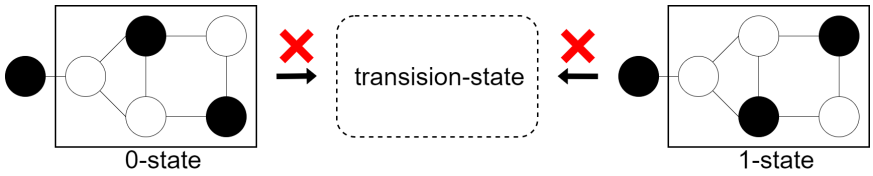


Fig.4 Blocking transitions depending on the state of the external piece

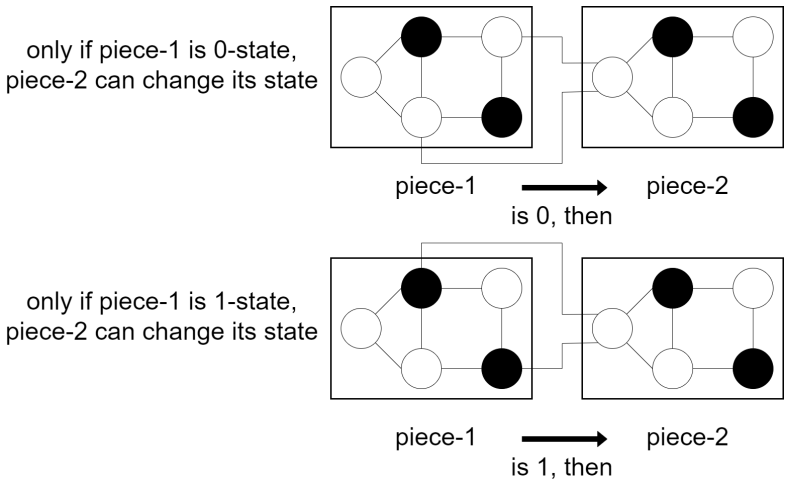


Fig.5 Connection of each piece