Agent motion planning with pull and push moves

Tadeu Zubaran, Marcus Ritt

Departamento de Informática Teorica, Instituto de Informática
Universidade Federal do Rio Grande do Sul, Porto Alegre, Brasil

Abstract. *Agent motion planning is a common task in artificial intelligence. One of the simplest scenarios considers the delivery of boxes to storage locations on a regular grid with obstacles. When the agent can only push boxes, we obtain the well-known, PSPACE-hard Sokoban puzzle [Culberson 1997]. In this paper we propose an exact solver for the scenario called* Pukoban *where the agent can push and pull boxes. The solver is, to the best of our knowledge, the first one proposed for Pukoban. It is based on the A* search algorithm with several problem-specific improvements. We evaluate its efficiency on* 100 *instances from the literature. Our algorithm is able to solve* 30 *instances exactly.*

1. Introduction

Pukoban is a game on an integer grid, where an agent (the warehouse keeper or robot) has to move boxes to designated storage locations. Each grid cell can contain a box, or an unmovable obstacle (or wall), which neither the agent nor a box can occupy, or a storage (or target) location, where a box must be placed to solve the game. Cells that do not contain a wall nor a box form the *free space*. The agent can push or pull a box one cell horizontally or vertically if the destination cell is free and he has enough space to do so. When the agent succeeds in placing every box at a storage locations the puzzle is solved. This simple set of rules can be deceiving as even a relatively small puzzle can be very hard to solve for both computers and humans alike, requiring that we use clever strategies to solve them. The goal of these puzzles is to study the solvability of abstract versions of typical motion planning situations, e.g., in robotics [Dor and Zwick 1999].

The reader can find an instance of a Pukoban puzzle in Figure 1. It shows one of the easiest and smallest test cases for our solver.

1.1. Other Versions of the Game

A slightly different and much more famous puzzle where the worker can only push the boxes is known as Sokoban. Schaeffer and Junghanns present a study of techniques to solve Sokoban puzzles [Schaeffer and Junghanns 2000] . When we attempt to solve a

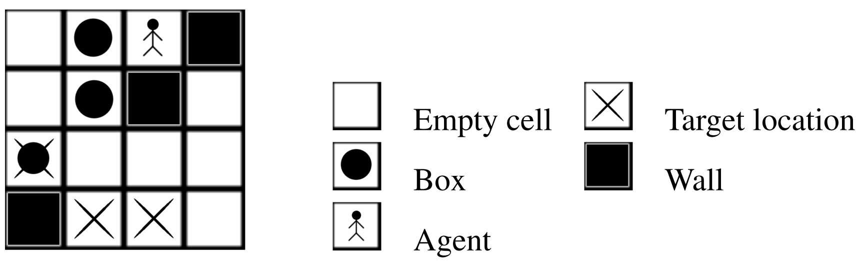


Figure 1. Example instance of Pukoban. The optimal solution needs 48 box movements.

Pukoban game either with a computer or by hand the successful strategies tend to be somewhat different from those successful in Sokoban, in spite of the striking similarities in their rule set.

There are many other versions of Sokoban-like games. Such versions include, for example, movable obstacles, allow the keeper to push up to k boxes, or even an unlimited number, require boxes to slide until hitting the next obstacle, or include boxes which occupy more than one cell. Most of these version have been proved to be NP-hard or PSPACE-complete [Demaine et al. 2003]. The NP-hardness of the Pukoban puzzle is, to the best of our knowledge, open.

While there is all this plethora of modifications of Sokoban there are other similar puzzles that have being explored in the literature and provided some insight on how to solve Pukoban puzzles such as Atomix [Hüffner et al. 2001].

2. Exact solution of Pukoban

Solving Pukoban is equivalent to finding the shortest path from the initial state to some solution state in the *state graph*. The vertices of this graph are the possible states of the game. Two states are joined by an arc, if there is a move transforming the first into the second one. In this work we define the number of box movements as the distance metric. One might be interested in minimising the number of movements of the worker itself, which is not considered by our approach.

2.1. Comparison with Sokoban

The rules for a Pukoban are quite similar to the Sokoban puzzle however they differ in key characteristics making the process of automatically solving a Pukoban game quite different from a Sokoban game.

First, and perhaps most importantly, a Sokoban search space graph is directed, making deadlocks possible. This is particularly important in man-made maps where most of the opening moves put the game in deadlock. Junghanns cleverly exploited this characteristic using a deadlock table that drastically reduces the branching factor – the number of possible moves in a given state, equal to the (out-)degree of the state in the state graph – of his search tree [Schaeffer and Junghanns 2000]. In Pukoban every move is reversible, and thus our search space graph is undirected and it is impossible to form a deadlock.

Solving Sokoban can be accelerated by using so-called tunnel macros. In Sokoban once the agent starts to push the box through a tunnel of width one, no target cells are in the tunnel, and there is no possibility to get to the other side of the tunnel, the agent must push the box all the way through the tunnel. In Pukoban there is no such limitation and the tunnel can conceivably be used as storage place for the box. Another key difference is that, even without considering deadlocks, the branching factor of Pukoban is usually bigger than that of Sokoban since there are more possibilities for the worker, leading to considerably larger search trees, and thus making the puzzle harder to solve.

2.2. Pukoban-specific Strategies

Our approach is based on the A* algorithm [Hart et al. 1968] applied to the state graph of the game to find a path of minimum cost from an origin state to a target state, along with

Algorithm 1 Pseudo code for the A* algorithm with *closed set*

```
open set contains only start node
closed set is empty
while open set is not empty do
    current node is the node with the cheapest total cost in open set
    remove current node from open set and put it in closed set
    if current node is target node then
        shortest path has been found: puzzle is solved
    else
        for each neighbour of current node do
            if current neighbour is not in open set nor in closed set then
                put neighbour in open set
            end if
        end for
    end if
end while
target node is unreachable from the start node
```

several improvements. A* is a modification of Dijkstra's shortest path algorithm using a heuristic which estimates the distance to the target to guide the exploration of the search space. A* has the same worst case complexity $O(m + n \log n)$ as Dijkstra's algorithm (when implemented using a Fibonacci heap), where n and m are the number of vertices and edges of the search graph. Observe that in our case the search graph has up to $\binom{rc}{b}$ vertices, for a map of dimension $r \times c$ and b boxes, which is exponential in the size of the input. Algorithm 1 shows the pseudo-code for the A* algorithm.

We call *current cost* of a node the cost to get from the start to the current node, *heuristic cost* the cost predicted by the heuristic to get from the current node to the target node, and *total cost* the sum of the current cost and the heuristic cost.

When we reach the target state, the current solution is guaranteed to be optimal for *admissible* heuristics. To be admissible the estimated distance to the target can never exceed the true distance. It is desirable that the heuristic is also *monotone*. A heuristic is monotone if for every pari of states x and y we have $h(x) \leq d(x, y) + h(y)$ where $h(x)$ is the estimated distance of state x to the target state, $d(x, y)$ is the actual distance to go from state x to state y. This means that it is impossible to decrease the *total cost* of a path between two nodes by adding another node in the path. As it will be seen in the following sections our total cost never decreases so all our heuristics are monotone. This helps A* because once A* visits a node (i.e. the node is the node with the lowest *total cost*) we can guarantee there will be no cheaper path to it so we can use a *closed set*, the set of nodes that do not need to be explored again improving the algorithm efficiency.

2.2.1. Distance Heuristic

We need a heuristic to guide the search of the A* algorithm. The quality of this heuristic usually has a significant effect on the overall performance of the algorithm. A simplistic yet computationally efficient approach is to compute the Manhattan distance of each box

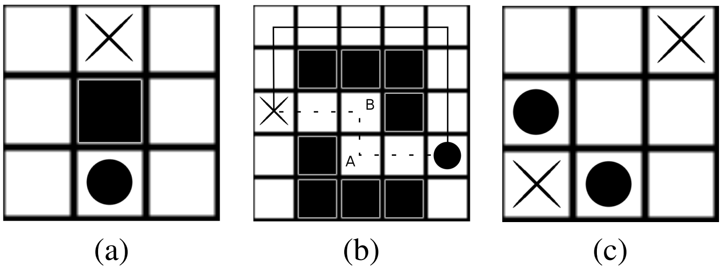


Figure 2. (a) The Manhattan distance underestimates the cost. (b) A path without enough space for the worker. (c) Multiple boxes concur for the same destination.

to the closest target cell. The sum of these values for all the boxes will give us a monotone, admissible heuristic. It is monotone because we do not decrease the total cost, if we are able to follow the path predicted by the heuristic. Otherwise the total cost will increase. The same argument applies to the following heuristics. If we compute the distance of each non-wall cell to its nearest target in a preprocessing step, we can compute the heuristic in $O(N)$, where N is the number of boxes.

We can improve the heuristic by taking into account the actual geometry of the map and use the distance to the closest target in free space. In Figure 2(a) the Manhattan distance to the target is two, yet the box can not be moved over the wall so a cost of four would be much more accurate. This technique gives us a more accurate yet still admissible and monotone heuristic. We introduce a small overhead in the preprocessing, but the cost of dynamically computing the heuristic remains $O(N)$.

Another improvement is to take into account that the box has to be either pushed or pulled by the worker so we can only take paths where there is space for the worker itself. In Figure 2(b) our previous heuristic would give us a cost of five for moving the box along the dashed line. But this is an impossible route because there is no space for the worker to move the box from A to B. Taking this into account we obtain a heuristic cost of nine for the path indicated by a continuous line. Similar to our previous improvement this introduces only a small overhead in the preprocessing but has no effect in the dynamic computing of the heuristic cost. We call this the *shortest distance* heuristic. Notice that it remains admissible and monotone, making it strictly better than the previous two.

We can further improve the heuristic taking into account that each target can be occupied by at most one box. In Figure 2(c) our current heuristic is two, but obviously we need at least four moves to bring both boxes to target positions. Thus the heuristic can be improved by considering only matchings of boxes and target positions. We can not allow an arbitrary matching since our heuristic can never overestimate the cost to guarantee its admissibility. Therefore we compute the distance of each of the non-wall cells to each of the targets. With that information we can compute the minimum matching of the distances of the boxes to the targets. The sum of the costs of this matching can be safely used as our heuristic keeping it monotone and admissible. We call this the *matching heuristic*.

A minimum matching in a bipartite graph (in our case the graph consisting of current and target positions) can be found in polynomial time using, for example, the Hungarian (or Kuhn-Munkres) algorithm [Kuhn 1955]. Like our previous improvements this introduces a small overhead in the preprocessing since we now have to compute the distance of each cell to each of the targets, but unlike our other improvements this also makes the dynamic computing of the heuristic more expensive because the minimum matching has to be calculated for each computed node of the search space.

2.2.2. Inertia

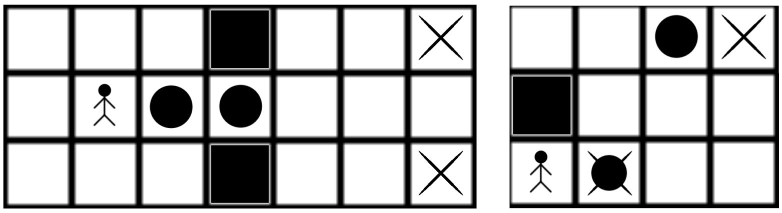
When a human solves an open map (either in a Sokoban or a Pukoban game) it is normal to see the same box being moved several times in a row. We use this characteristic to break ties between states of the same total cost, giving preference to states that keep the agent moving the same box. This was discussed by [Schaeffer and Junghanns 2000] when solving Sokoban puzzles and translates quite well to our case.

2.2.3. Choke Points

Several of the Sokoban specific maps have a room (a connected region of cells) where all the targets are [Schaeffer and Junghanns 2000]. In such a situation further optimisations are possible. We define a *choke point* as a cells that separates targets from boxes, i.e., when a choke point is blocked, no box can reach any target. If we find multiple choke points we choose the one that minimises the size of the room containing the targets. We call the choke point and all the cells beyond this point in the direction of the targets *inside* the room and the other cells *outside*. Choke points can be found in a preprocessing step by applying a direct test of the definition above. The following improvements are only possible when we find a choke point.

Prefer Box Closest To Target An improvement based on the strategy that humans tend to use when solving open maps is to attempt to move boxes that are already near a target first. When there are nodes tied in cost, we prefer to explore first those at positions inside the target room. If there is no such box accessible, we prefer the box closest to the choke point. Both inertia and this improvement usually point to the same node to be explored first. When they differ the box closest to the target has priority over inertia, since this may help to remove obstacles for the later boxes.

Dynamic Computation of Minimum Matchings If we have a choke point sometimes the minimum matching can be updated using the matching of the previous state. If we move a box that was outside the target room at position c in state s to position c' in a state s' that still is outside, then the new heuristic is given by $h(s') = h(s) + d(c') - d(c)$, where $h(x)$ is the heuristic cost of the state x and $d(y)$ is the distance (as defined when we calculate the heuristic) of the position y to the choke point. To show this we point out that the heuristic cost of any box outside with regard to any of the targets $t \in T$ is given by $d_t(a) = d_c(a) + d_t(c)$, where $d_x(y)$ is the heuristic distance from y to x and T the set of all targets. By hypothesis the moved box was outside, so only $d_c(a)$ changes, meaning the distance of the box to all the targets changes by the same amount, so the minimum matching of the previous state is also the minimum matching of the new state, but with a cost increased by $d(c') - d(c)$. Since most of the search is performed moving outside boxes and we can compute the heuristic in constant time in such cases, this improves the performance considerably.



(a) (b)

Figure 3. (a) Example of a clog: The worker has to move a box away from the targets to be able to solve the puzzle. (b) Example of an ineffective clog: The agent can not move when the condition of the clog is satisfied.

Clogs The state graph of a Pukoban game is bidirectional, therefore we do not have deadlocks but during the process of solving the game sometimes we form some structures that cannot be undone without increasing the total cost. We call these structures *clogs*. Clogs can be formed by two or more boxes. Figure 3(a) gives an example of a two-box clog.

Most clogs depend on the position of the worker as well as the position of the boxes that form the clog. In the example above we would not have a clog (meaning we would not underestimate the heuristic with certainty) if the worker was inside the room with the targets. A clog is completely defined by the position of the boxes that form the clog and a set of positions where the worker can be. We can calculate clogs in the preprocessing stage by attempting to solve the Pukoban puzzle with a limited number of boxes without increasing the heuristic.

Observe that since the matching of a subset of the boxes to the targets in the optimal solution is unknown, we can not just compute the minimal matching of the boxes to the targets to decide if they form a clog. However, in the common situation that there exists a choke point, and the candidate boxes are outside the target room, the matching of boxes to targets is irrelevant, and we can safely identify clogs. We further limit the preprocessing to computing clogs with a small number of boxes since computing all clogs with the any number of boxes would mean solving the original Pukoban problem (along with several others), rendering this process useless.

We can increase the heuristic cost when we detect a clog in some state. We choose a conservative approach of increasing the heuristic by the minimal possible increase of two, if we find at least one clog. The process of using clogs to improve the heuristic introduces an overhead to find the clogs when preprocessing and during the execution of the A*, because we need to test if any clogs are present at each new state. This is true even if we use the choke point optimisation where we can use the previous heuristic to compute the new heuristic faster since any move of a box may form or destroy a clog.

We chose to look for two-box clogs only. This is already a costly preprocessing and usually yields between ten and 500 clogs. We further trim the types of clogs we will be looking for by eliminating those that either limit the position of the worker too much (i.e. the worker can not move more than a few cells) or consist of very distant boxes, since such patterns are very uncommon. Figure 3(b) shows an example of the former situation. Since the agent can not move if we satisfy the condition of the clog, we can remove it without increasing the cost of the search.

3. Notes on the Implementation

A small memory footprint of the nodes is important for a good performance of the A* search. We store the static geometry of the maze (i.e. position of walls, targets and free spaces) only once in a two-dimensional array. The worker's position is stored as a pair of 16-bit integers, while the position of the boxes and the targets is stored as a set of such pairs. The program also computes and stores the cells reachable by the agent without moving a box in a bitmap. Using this representation a node in the search space occupies only $2(n + 1) + rc/8$ bytes, for n boxes and a map of size $r \times c$.

If the map contains a choke point, we store another bitmap indicating the cells inside the target room and position of the choke point. The clogs found during preprocessing are stored in a list. Each element of the list contains the set of the boxes forming the clog and a bitmap of the agent's positions that make the clog effective. If we use the heuristic without the minimum matching we store the distance of each position to the closest target in a two-dimensional array; if we use the minimum matching then we store a three-dimensional array with the distance between every cell and every target.

The open and the closed set are two key data structures of A* that must be implemented efficiently (see Algorithm 1). The closed set has to support addition of a new element and a test if some element is already contained. We chose a hash table to represent the open set, since it can execute both operations in amortised constant time provided we have a collision-free hash function. Due to the large number of states, computing such a function can be very expensive so we made a compromise between the number of hash conflicts and the cost to compute the hash function. The function is given by $H(s) = \sum_{1 \leq i \leq n} rn2^i c_i + ir_i$ for n boxes at positions (r_i, c_i), $1 \leq i \leq n$. The open set can be implemented as a priority queue, which supports the removal of the element with the lowest total cost and the update of the distance of an element already contained in the queue, when we have found a shorter path to the node in question. Our implementation combines the priority queue with an additional hash table to efficiently find the states already stored in the queue. This way an update has cost logarithmic in the number of open states, the extraction of the smallest element has constant amortized time.

4. Experimental Results

We have implemented our algorithm in C++. (A preliminary version of the solver implemented by our group is available at [Jurkovski 2010].) It has been compiled using the GNU C++ 4.4.1 compiler with the "-O3" option. We used a PC with an Intel Core i7 930 processor with 12 GB of main memory for the experiments. The time limit for each test was 3600 seconds. All times are wall clock time.

We tested our algorithm on two quite distinct sets of test cases. The first set consists of small, densely populated maps designed specifically for the Pukoban game [Clercq]. Humans tend to have difficulty solving these maps, while computers can solve them with relative ease since the search space tends to be small. The second set of maps consists of much bigger maps, which were designed for Sokoban and tend to have much more free space between the boxes. Humans typically solve Pukoban games of these maps with relative ease close to optimality. The search space grows rapidly with the size of the map, and the free space leads to lots of equivalent moves, so these are challenging maps for a solver.

We tested the shortest distance heuristic and the matching heuristic with three groups of improvements: a *basic* set consisting only of box ordering, a *medium* set adding inertia, choke points and giving preference to boxes closer to the destination, and a *complete* set further adding clogs. Due to space limitations, we present only five of the 12 experiments. Table 1 presents the results for the Pukoban instances with the shortest distance heuristic with basic improvements, and the matching heuristic with complete improvements, respectively. Tables 2, 3, and 4 present the results for the XSokoban instances using the matching heuristic and basic, medium, and complete improvements. In all tables column "M" reports the number of moves if an optimal solution has been found or "-" otherwise, column "T" reports the execution time in seconds, column "V" reports the number of visited nodes, column "Nodes" reports the total number of nodes, and column "BF" reports the branching factor.

The branching factor depends on the characteristics of the instance (e.g. the number and placement of obstacles) and varies between 3 and 30. We can observe small fluctuations for different heuristics, which can explained by the different order in which states are explored. The larger instances have an average branching factor of about 12.3, which is larger than the typical branching factor of 10 for Sokoban and 7 for Atomix [Hüffner et al. 2001], showing that we have to explore more states to find an optimal solution.

All ten of the smaller Pukoban instances can solved in less than five seconds even with the simplest heuristic. Actually, the better heuristic almost doubles the execution time, since it is not able to amortize its higher computational cost by pruning reducing the already small search space. Looking at the XSokoban instances, we can see positive effects of better heuristics. In our experiments, the simple distance heuristics could not solve any instance. Using the matching heuristic, but only box ordering, we are able to solve 15 of the 90 XSokoban instances. Adding inertia and choke points the solution time for the already solved puzzles drops significantly, but we are able to solve only two more puzzles.

Finally, when adding clogs, the number of solved instances increases to 20. Of all the improvements, clogs have the most significant impact on the solution of the puzzle, except the quality of the heuristic. When introducing clogs, the number of visited nodes and the execution time drops by 10% for puzzles already solved with medium improvements. To evaluate the quality of the our solver, we can compare it to Sokoban where the currently best implementation is able to solve 57 of 90 instances [Schaeffer and Junghanns 2000].

5. Conclusions and Future Work

We have proposed an exact solver for Pukoban puzzles. It is able to solve 30 of 100 challenging instances. The most important characteristics which make this possible are a precise distance heuristic, which can be computed efficiently, and the detection of patterns of boxes which increase the estimated distance. We are currently evaluating the quality of the proposed algorithm as an approximate solver, since the heuristic usually is very close to the optimal solution, and most of the time is spent in proving optimality of a solution found early in the search.

Since the implementation has a low memory footprint there was no need to use more sophisticated algorithms to save memory such as IDA* [Korf 1985]. We intend to study the behaviour of IDA* to solve the larger instances. Other future improvements are more effi-

Table 1. Results for Pukoban instances. Left: Shortest distance heuristic. Right: Matching heuristic, box ordering, inertia, choke points and clogs.

Map	M	T	V	Nodes	BF
1	48	0	433	1255	2.9
2	51	0.03	1056	3464	3.3
3	51	0.02	1028	3332	3.2
4	62	0.01	1176	3473	3.0
5	59	0.02	850	2742	3.2
6	80	0.27	11808	37431	3.2
7	114	0.17	8865	26544	3.0
8	85	0.14	6645	21200	3.2
9	94	1.22	31632	99005	3.1
10	173	1.82	41797	130804	3.1

Map	M	T	V	Nodes	BF
1	48	0.01	431	1247	2.9
2	51	0.03	1014	3367	3.3
3	51	0.04	879	2822	3.2
4	62	0.03	1127	3352	3.0
5	59	0.02	802	2608	3.3
6	80	0.59	11642	37040	3.2
7	114	0.38	8792	26338	3.0
8	85	0.37	6637	21188	3.2
9	94	2.63	31610	98961	3.1
10	173	4.03	41746	130721	3.1

cient data structures, such as the data structure proposed by Hüffner [Hüffner et al. 2001] for a better priority queue management, and the evaluation of other search space pruning techniques. Pukoban has a lot of potential for new discoveries applying domain knowledge for finding and exploring patterns such as clogs, which can improve the distance estimate to the solution in specific situations. As an example, we may be able to improve the heuristic in the presence of multiple clogs.

References

Clercq, D. D. Pukoban. `http://puzzles.net23.net/pukoban.htm`.

Culberson, J. C. (1997). Sokoban is PSPACE-complete. Technical report, University of Alberta, 97-02.

Demaine, E. D., Demaine, M. L., Hoffmann, M., and O'Rourke, J. (2003). Pushing blocks is hard. *Computational Geometry*, 26:21–36.

Dor, D. and Zwick, U. (1999). Sokoban and other motion planning problems. *Computational Geometry*, 13:215–228.

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

Hüffner, F., Edelkamp, S., Fernau, H., and Niedermeier, R. (2001). Finding optimal solutions to Atomix. In *Advances in artifical intelligence*, volume LNCS 2174, pages 229–243.

Jurkovski, B. (2010). Pukoban solver. Available: `https://github.com/bjurkovski/PukobanSolver`.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109.

Kuhn, H. W. (1955). The hungarian method for the assignment problem. *Naval Research Logistic Quarterly*, 2:83–97.

Schaeffer, J. and Junghanns, A. (2000). Sokoban: Enhanceing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129(1–2):219–251.

Table 2. Results for Xsokoban instances, using the matching heuristic and inertia.

Map	M	T	V	Nodes	BF	Map	M	T	V	Nodes	BF
1	87	0	125	1053	8.4	46	-	+	324987	6272285	19.3
2	117	2	6900	44840	6.5	47	-	+	413292	4143255	10.0
3	128	148	161348	1558977	9.7	48	-	+	245623	3317753	13.5
4	-	+	747745	4650435	6.2	49	-	+	420263	4485298	10.7
5	131	821	411147	4047201	9.8	50	-	+	452012	5359271	11.9
6	-	+	753529	5951821	7.9	51	-	+	564492	9591362	17.0
7	78	0	97	1328	13.7	52	-	+	404374	4507530	11.1
8	210	1	296	6242	21.1	53	-	+	677418	10060146	14.9
9	-	+	794601	8924617	11.2	54	-	+	434991	5077483	11.7
10	-	+	171838	2536990	14.8	55	-	+	677916	7932238	11.7
11	-	+	558184	7100514	12.7	56	179	45	40727	460398	11.3
12	-	+	247054	6862772	27.8	57	-	+	305656	6121244	20.0
13	-	+	438962	4910516	11.2	58	-	+	901254	6503799	7.2
14	-	+	306240	4010911	13.1	59	-	+	490509	5963677	12.2
15	-	+	427873	5460235	12.8	60	-	+	369231	6853194	18.6
16	-	+	580914	5332977	9.2	61	-	+	399899	5121082	12.8
17	-	+	1498085	11911426	8.0	62	-	+	310049	4681194	15.1
18	-	+	447757	4940984	11.0	63	-	+	447377	7489930	16.7
19	-	+	624368	8875012	14.2	64	-	+	390168	4629855	11.9
20	-	+	406655	5086489	12.5	65	-	+	486078	5495624	11.3
21	-	+	589015	7340251	12.5	66	-	+	797247	7643583	9.6
22	-	+	274599	2905952	10.6	67	-	+	870596	8129591	9.3
23	-	+	779940	6421782	8.2	68	-	+	503990	6613114	13.1
24	-	+	130118	2042980	15.7	69	-	+	731776	7651692	10.5
25	-	+	288138	4554366	15.8	70	-	+	578488	7525758	13.0
26	-	+	598874	5528350	9.2	71	-	+	587682	6803117	11.6
27	-	+	292889	3129733	10.7	72	-	+	600192	6305527	10.5
28	-	+	380271	5047256	13.3	73	-	+	1130097	13838485	12.2
29	-	+	1248308	5169734	4.1	74	-	+	758917	8324322	11.0
30	-	+	688126	4320733	6.3	75	-	+	463823	4273020	9.2
31	-	+	413349	3839136	9.3	76	-	+	364254	4775723	13.1
32	-	+	303835	4152244	13.7	77	-	+	645304	5506194	8.5
33	-	+	715774	5966822	8.3	78	126	0	131	2233	17.0
34	-	+	410182	7671479	18.7	79	164	0	166	2994	18.0
35	-	+	293196	3789752	12.9	80	-	+	941829	7539384	8.0
36	345	3311	968816	9929790	10.2	81	167	0	179	2967	16.6
37	-	+	462132	3888028	8.4	82	133	28	33199	361712	10.9
38	29	0	361	3631	10.1	83	194	3578	1379790	13222972	9.6
39	-	+	318771	4015157	12.6	84	141	0	201	3901	19.4
40	-	+	538140	8134070	15.1	85	-	+	270608	5128451	19.0
41	-	+	1148548	6977961	6.1	86	-	+	681512	8624364	12.7
42	-	+	207050	3971964	19.2	87	-	+	704518	7947805	11.3
43	-	+	844737	9124619	10.8	88	-	+	244461	3510997	14.4
44	-	+	1003579	9559526	9.5	89	-	+	371842	4316562	11.6
45	-	+	473525	5203336	11.0	90	-	+	611637	4573388	7.5

M : Moves. A “-” indicates that the optimal solution has not been found.

T: Time in seconds. A “+” indicates a time exceeding 3600.

V: Number of visited nodes.

BF: Branching factor.

Table 3. Results for Xsokoban instances, using the matching heuristic, box ordering, inertia and choke points.

Map	M	T	V	Nodes	BF	Map	M	T	V	Nodes	BF
1	87	0	125	1101	8.8	46	-	+	325529	6281460	19.3
2	117	0	2166	25571	11.8	47	-	+	414447	4153623	10.0
3	-	+	984518	10759268	10.9	48	-	+	245533	3316770	13.5
4	-	+	738609	10178233	13.8	49	-	+	423930	4526595	10.7
5	131	2	14042	80189	5.7	50	-	+	449481	5324368	11.8
6	-	+	901610	8641494	9.6	51	-	+	565276	9602237	17.0
7	78	0	97	1328	13.7	52	-	+	679712	9532711	14.0
8	210	1	296	6242	21.1	53	-	+	679179	10090817	14.9
9	-	+	998126	11646397	11.7	54	-	+	436405	5094118	11.7
10	-	+	177113	2631903	14.9	55	-	+	678934	7942215	11.7
11	191	1	4726	48610	10.3	56	179	43	40727	460398	11.3
12	-	+	267024	7448642	27.9	57	-	+	312557	6246182	20.0
13	-	+	455884	5074505	11.1	58	-	+	928279	6684635	7.2
14	-	+	323762	4229232	13.1	59	-	+	520410	6418169	12.3
15	-	+	470328	6042569	12.8	60	-	+	373725	6941073	18.6
16	-	+	584752	5360521	9.2	61	-	+	412022	5232583	12.7
17	-	+	1508163	11996694	8.0	62	235	0	242	3691	15.3
18	-	+	450441	4978555	11.1	63	-	+	456934	7697593	16.8
19	-	+	634292	8983881	14.2	64	-	+	387411	5628970	14.5
20	-	+	413107	5166390	12.5	65	-	+	494145	5589309	11.3
21	-	+	586851	7310533	12.5	66	-	+	815513	7818883	9.6
22	-	+	278067	2949167	10.6	67	-	+	898367	8327960	9.3
23	272	3	8264	71927	8.7	68	-	+	517425	6768557	13.1
24	-	+	129292	2031646	15.7	69	-	+	743286	7775347	10.5
25	-	+	224291	2915515	13.0	70	-	+	585291	7622824	13.0
26	-	+	683708	6254966	9.1	71	-	+	596014	6919532	11.6
27	-	+	653057	4767663	7.3	72	-	+	605611	6365504	10.5
28	-	+	396800	5182789	13.1	73	-	+	1156948	14141886	12.2
29	-	+	1238633	5130540	4.1	74	-	+	770812	8442975	11.0
30	-	+	685263	4302399	6.3	75	-	+	457082	4201340	9.2
31	-	+	414745	3854533	9.3	76	-	+	350637	4581203	13.1
32	-	+	308531	4236095	13.7	77	-	+	633572	5374093	8.5
33	-	+	718333	5990458	8.3	78	126	0	131	2233	17.0
34	-	+	413940	7734000	18.7	79	164	0	166	2994	18.0
35	-	+	294234	3807764	12.9	80	-	+	1278263	9673825	7.6
36	345	3288	968816	9929790	10.2	81	167	0	179	2967	16.6
37	-	+	464060	3906289	8.4	82	133	3	12149	114172	9.4
38	29	0	361	3631	10.1	83	194	1145	648937	5599484	8.6
39	-	+	447187	5768782	12.9	84	141	0	201	3901	19.4
40	-	+	419124	5144530	12.3	85	-	+	268697	5088022	18.9
41	-	+	1036220	11304848	10.9	86	-	+	767727	10232581	13.3
42	-	+	207027	3971594	19.2	87	-	+	709049	7995999	11.3
43	-	+	845884	9136733	10.8	88	-	+	242607	3484913	14.4
44	-	+	1005276	9580484	9.5	89	-	+	369655	4292151	11.6
45	-	+	591382	6713599	11.4	90	-	+	616302	4602935	7.5

M : Moves. A “-” indicates that the optimal solution has not been found.
T: Time in seconds. A “+” indicates a time exceeding 3600.
V: Number of visited nodes.
BF: Branching factor.

Table 4. Results for Xsokoban instances, using the matching heuristic, box ordering, inertia, choke points and clogs.

Map	M	T	V	Nodes	BF	Map	M	T	V	Nodes	BF
1	87	0	88	849	9.6	46	-	+	511667	7546028	14.7
2	117	1	2089	25493	12.2	47	-	+	370019	3446207	9.3
3	128	1	12072	58747	4.9	48	-	+	223040	3136806	14.1
4	327	0	328	9823	29.9	49	-	+	405296	4513823	11.1
5	131	2	13728	78143	5.7	50	-	+	417800	5075133	12.1
6	-	+	557734	6802452	12.2	51	-	+	559787	9527692	17.0
7	78	0	97	1328	13.7	52	-	+	664429	9578048	14.4
8	210	1	296	6242	21.1	53	-	+	685347	10207725	14.9
9	-	+	533730	5579917	10.5	54	-	+	430857	5035762	11.7
10	-	+	172697	2549974	14.8	55	-	+	676478	7918272	11.7
11	191	1	4555	47490	10.4	56	179	43	40727	460398	11.3
12	-	+	258335	7186237	27.8	57	-	+	311050	6210172	20.0
13	-	+	427501	4803970	11.2	58	-	+	904296	6527903	7.2
14	-	+	308726	4037505	13.0	59	-	+	484496	5882935	12.1
15	-	+	441718	5663094	12.8	60	-	+	362076	6716824	18.6
16	-	+	562795	5180993	9.2	61	-	+	344446	4495913	13.1
17	-	+	1502845	11944752	7.9	62	235	0	236	3662	15.5
18	-	+	452349	5004612	11.1	63	-	+	440801	7333880	16.6
19	-	+	613026	8772684	14.3	64	-	+	377204	5486005	14.5
20	-	+	414637	5184949	12.5	65	-	+	491234	5552317	11.3
21	-	+	602391	7550315	12.5	66	-	+	814817	7811358	9.6
22	-	+	271109	2868611	10.6	67	-	+	898984	8331686	9.3
23	272	3	7739	68856	8.9	68	-	+	518070	6777196	13.1
24	-	+	131117	2057483	15.7	69	-	+	745022	7789062	10.5
25	-	+	247855	3235373	13.1	70	-	+	585147	7620891	13.0
26	-	+	666969	7893205	11.8	71	-	+	593326	6880524	11.6
27	-	+	443317	3781240	8.5	72	-	+	604484	6349325	10.5
28	-	+	403603	5241978	13.0	73	-	+	1153156	14099720	12.2
29	-	+	1258655	5210841	4.1	74	-	+	769904	8433590	11.0
30	-	+	713702	4482867	6.3	75	-	+	465790	4295976	9.2
31	-	+	412396	4044442	9.8	76	-	+	365590	4796145	13.1
32	-	+	306827	4204944	13.7	77	-	+	652735	5585089	8.6
33	-	+	696681	5845747	8.4	78	126	0	131	2233	17.0
34	-	+	404778	7582671	18.7	79	164	0	166	2994	18.0
35	-	+	270892	3281111	12.1	80	219	3278	936075	7492699	8.0
36	345	3105	897979	9493586	10.6	81	167	0	178	2963	16.6
37	-	+	459437	3921612	8.5	82	133	1	4585	41826	9.1
38	29	0	359	3623	10.1	83	194	16	74091	583068	7.9
39	-	+	450074	5802580	12.9	84	141	0	157	3508	22.3
40	-	+	415672	5097355	12.3	85	-	+	277516	5266711	19.0
41	-	+	1042130	11366048	10.9	86	-	+	740512	10172732	13.7
42	-	+	222883	4736412	21.3	87	-	+	717532	8108724	11.3
43	-	+	782197	9579875	12.2	88	-	+	252170	3628606	14.4
44	-	+	921674	9321561	10.1	89	-	+	380273	4399666	11.6
45	-	+	335943	3580918	10.7	90	-	+	636812	4729089	7.4

M : Moves. A "-" indicates that the optimal solution has not been found.

T: Time in seconds. A "+" indicates a time exceeding 3600.

V: Number of visited nodes.

BF: Branching factor.