Parallelizing Classical Planning: Critical Path Heuristics on the GPU

Markus Fritzsche[1], **David Speck**[2], **Daniel Gnad**[3,1], **Simon Ståhlberg**[4]

[1]Linköping University, Sweden
[2]University of Basel, Switzerland
[3]Heidelberg University, Germany
[4]RWTH Aachen University, Germany
markus.fritzsche@liu.se, davidjakob.speck@unibas.ch, daniel.gnad@uni-heidelberg.de, simon.stahlberg@gmail.com

Abstract

Despite the tremendous capabilities of modern hardware in performing parallel computations, all major classical planners are limited to single-threaded execution on the CPU. We show how the critical path heuristic h^m, commonly used in classical planning, can be parallelized and computed on a GPU. To that end, we construct a directed hypergraph, where nodes represent sets of atoms, associated with their reachability costs, and actions define weighted hyperedges. Iteratively performing convolutions on this hypergraph until a fixed point is reached allows us to efficiently compute h^m on the GPU. Furthermore, it enables batching, so we can compute the heuristic in parallel for multiple states. Our approach naturally supports multiple cost functions, allowing efficient computation of cost partitioning for h^m. We demonstrate experimentally that the GPU-based computation of h^m can achieve speedups of several orders of magnitude over the traditional computation on a CPU.

Code — https://github.com/mrlab-ai/downward-h2
Datasets — https://doi.org/10.5281/zenodo.20423666

Introduction

Modern hardware has undergone a remarkable transformation over the past decade, with GPUs and multicore CPUs enabling massively parallel computation across a wide range of domains. Yet classical planning, one of the foundational areas of symbolic AI, has largely failed to capitalize on these advances. All major planning frameworks operate on a single CPU core, executing heuristic search in a strictly sequential fashion. This limitation was evident at the 2023 International Planning Competition (IPC), which offered no multicore track and evaluated planners exclusively on single-threaded performance (Taitler et al. 2024).

Heuristic search, the dominant paradigm in classical planning (Bonet and Geffner 2001; Hoffmann and Nebel 2001), poses a fundamental challenge for parallelization (Kishimoto, Fukunaga, and Botea 2013). Open and closed lists must be synchronized across processes, and many heuristics are deliberately designed to be lightweight, making the overhead of parallel execution prohibitive. A natural response to this is to invest in more informative (and more computationally demanding) heuristics, whose parallelization is more likely to pay off. The critical path heuristic h^m (Haslum and Geffner 2000) is a compelling candidate: it is admissible, increasingly informative as m grows, and its computation involves many independent updates that are amenable to parallel execution. However, its exponential scaling with m limits practical use largely to $m = 1$ (i.e., $h^{\max}$) or to $m = 2$ in preprocessing (Alcázar and Torralba 2015).

We show how h^m can be efficiently computed on the GPU. Our approach represents the heuristic computation as a series of convolutions on a directed hypergraph, akin to graph convolutional networks (Kipf and Welling 2017), in which nodes correspond to sets of ground atoms labeled with reachability costs and actions define weighted hyperedges. Iteratively applying these convolutions until a fixed point is reached yields an admissible heuristic that correspond exactly to h^m. This formulation is closely related to the generalized Bellman-Ford algorithm (Ford 1956; Bellman 1958), but with all updates performed in parallel, exploiting the massive throughput of modern GPUs. The hypergraph representation naturally extends to support batching, i.e., computing heuristic values for multiple states simultaneously, and to support multiple cost functions, enabling efficient GPU-based computation of cost partitioning for h^m.

Our work builds on the connection between h^m and *hyper-abstractions* (Steinmetz and Torralba 2019), which represent abstract states and transitions via hypergraph structures. While this connection has been established theoretically, no prior experimental evaluation of h^m in this setting exists. GPU acceleration of state-space search in planning has been explored before (Edelkamp and Sulewski 2009; Sulewski, Edelkamp, and Kissmann 2011). The most closely related empirical work is that of Shipovalov and Pryanichnikov (2020), who implement GBFS and $h^{\max}$ using CUDA, demonstrating that GPU acceleration can yield substantial speedups in planning. Our work extends this line of research by targeting the richer h^m family and exploiting the full structure of the hypergraph formulation to enable batching and cost partitioning.

We integrate our GPU implementation into Fast Downward (Helmert 2006) and focus on evaluating h^2 on standard IPC benchmarks. Our experiments demonstrate speedups of up to several orders of magnitude over the sequential CPU

baseline, establishing GPU-accelerated h^m computation as a practically viable and theoretically well-grounded approach to parallelizing classical planning.

Background

Classical Planning. A STRIPS planning task is a tuple $P = \langle \mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ (Fikes and Nilsson 1971), where $\mathcal{V}$ is a set of propositional variables, also called atoms, states are subsets $s \subseteq \mathcal{V}$, $\mathcal{A}$ is a set of actions, $\mathcal{I} \subseteq \mathcal{V}$ is the initial state, and $\mathcal{G} \subseteq \mathcal{V}$ is the goal. We denote the set of all states as S. For each action $a \in \mathcal{A}$, $\mathrm{pre}(a)$, $\mathrm{del}(a)$, $\mathrm{add}(a)$, and $\mathrm{cost}(a)$ denote its precondition, delete list, add list, and cost. An action a is applicable in s if $\mathrm{pre}(a) \subseteq s$, in which case applying a yields $a \oplus s = (s \setminus \mathrm{del}(a)) \cup \mathrm{add}(a)$.

A solution to a state s of P is a sequence of actions $a_1, \ldots, a_n$ such that, starting from $s_0 = s$, each action a_i is applicable in s_{i-1} and produces $s_i = a_i \oplus s_{i-1}$ for $i = 1, \ldots, n$, with $\mathcal{G} \subseteq s_n$. A solution for P is a solution for its initial state $\mathcal{I}$. Its cost is $\sum_{i=1}^{n} \mathrm{cost}(a_i)$. We denote the cost of an optimal solution to a state s by $h^*(s)$.

A heuristic is a function $h : S \mapsto \mathbb{N}$ mapping states to goal-distance estimates (Hart, Nilsson, and Raphael 1968). It is admissible if $h(s) \leq h^*(s)$ for all states $s \in S$.

Critical Path Heuristic. We restate only the ingredients of h^m needed later. In this subsection, the argument s of $h^m(s)$ ranges over sets of atoms obtained by regression from forward search states. We follow the definition of Haslum, Bonet, and Geffner (2005). Given a task P, let $R(P)$ denote the set of transitions (s, a, s') such that regressing s via action a yields s', that is, $s \cap \mathrm{del}(a) = \emptyset$ and $s' = (s \setminus \mathrm{add}(a)) \cup \mathrm{pre}(a)$. The h^m heuristic is defined as:

$$h^m(s) = \begin{cases} 0 & \text{if } s \subseteq \mathcal{I} \\ \min_{(s,a,s') \in R(P)} h^m(s') + \mathrm{cost}(a) & \text{if } |s| \leq m \\ \max_{s' \in \mathcal{P}_{\leq m}(s)} h^m(s') & \text{otherwise} \end{cases}$$

where $\mathcal{P}_{\leq m}$ denotes the powerset restricted to subsets of size at most m. The function $h^m(s)$, as defined above, is an admissible heuristic (Haslum, Bonet, and Geffner 2005). The three branches correspond to a base case, a regression case, and a decomposition case: satisfied subgoal sets get cost 0; sets of size at most m consider all single-step regressions and keep the cheapest one; larger sets are approximated by the most expensive subset of size at most m.

This recurrence can be solved by dynamic programming over all atom sets of size at most m, e.g., via a generalized Bellman-Ford algorithm (Haslum 2006), which we build our h^m computation on and which serves as our CPU baseline.

Example. Let $m = 2$, $\{x, y, p, q, r\} \subseteq \mathcal{V}$ be propositions of the task and consider the set $t = \{p, q, r\}$. Then, we decompose t by $h^2(t) = \max_{s' \in \mathcal{P}_{\leq 2}(t)} h^2(s')$, i.e., the maximum over all subsets of t of size at most 2. Now suppose an action a has $\mathrm{pre}(a) = \{x, y\}$, $\mathrm{add}(a) = \{p, q, r\}$, and $\mathrm{del}(a) = \emptyset$. Then for each such subset s', regressing s' over a yields $\{x, y\}$, and thus $h^2(s') \leq h^2(\{x, y\}) + \mathrm{cost}(a)$.

```
1: all_vals = index_select(
     input=vertices, dim=0,
     index=value_indices)
2: max_vals = index_reduce(
     input=all_vals, dim=0,
     index=max_indices, source=all_vals,
     reduce="amax", include_self=False)
3: max_vals = max_vals + edge_weights
4: vertices = index_reduce(
     input=vertices, dim=0,
     index=min_indices, source=max_vals,
     reduce="amin", include_self=True)
```

Figure 1: PyTorch implementation of the convolution operator. The tensor `vertices` stores the current vertex values, `value_indices` encodes tails, `max_indices` and `min_indices` encode the reductions, and `edge_weights` stores the hyperedge weights.

Hypergraph Representation

We encode the dynamic program to compute h^m as a fixed directed hypergraph $G = (X, E)$ whose vertex labels are updated in parallel (Gallo, Longo, and Pallottino 1993). The vertex set $X = \mathcal{P}_{\leq m}(\mathcal{V})$ consists of atom sets of size at most m. These vertices are exactly the table entries of the generalized Bellman-Ford algorithm, and each stores the current estimate for achieving its atom set. Each edge $(T, H) \in E$ represents a transition $(s, a, s') \in R(P)$ and has weight $\mathrm{weight}((T, H)) = \mathrm{cost}(a)$. The tail T and head H are defined as:

$$T = \mathcal{P}_{\leq m}(s'), \quad H = \{v\}, v \in \mathcal{P}_{\leq m}(s).$$

In other words, a hyperedge uses the current estimates of the subsets in T to propose an update for the single head vertex in H. Each transition in $R(P)$ may therefore yield multiple hyperedges, one for each choice of $v \in \mathcal{P}_{\leq m}(s)$.

Note that the hypergraph depends only on the planning task and on m. Different search states do not change the graph itself; they only change the initial labels assigned to the vertices before the fixpoint computation starts.

Example. Continuing the example with $m = 2$, the regression transition $(\{p, q, r\}, a, \{x, y\})$ induces the tail $T = \{\emptyset, \{x\}, \{y\}, \{x, y\}\}$. Choosing the head vertex $v = \{p, q\}$ yields the hyperedge $(T, \{\{p, q\}\})$. Choosing instead $v = \{p, r\}$ or $v = \{r\}$ similarly yields additional hyperedges, each with the same tail but a different head. This is the graph-level analogue of applying the regression case to different subsets of the larger subgoal set.

Hypergraph Convolution

With the hypergraph fixed, one iteration of the dynamic program becomes a uniform operator over all hyperedges. This is the operation we map to the GPU.

Let $d^t : X \to \mathbb{N} \cup \{\infty\}$ denote the vertex labels after t convolution rounds. One convolution round first computes, for every hyperedge $e = (T, \{v\})$,

$$u_e^t = \max_{x \in T} d^t(x) + \mathrm{weight}(e),$$

and then updates the head vertex by

$$d^{t+1}(v) = \min\left(d^t(v), \min_{e=(T,\{v\})\in E} u_e^t\right).$$

Given a state s, we initialize $d^0(v) = 0$ if $v \subseteq s$ and $d^0(v) = \infty$ otherwise. This realizes the base case of the h^m recurrence. Let d^* denote the converged vertex labeling. The heuristic value is then obtained as $\max_{g\in\mathcal{P}_{\leq m}(\mathcal{G})} d^*(g)$. This graph readout is correct: the hypergraph above is the h^m hypergraph of Steinmetz and Torralba (2019), so the converged labeling satisfies $d^*(v) = h^m(v)$ for every $v \in X$, and the maximum over size-at-most-m subsets of $\mathcal{G}$ recovers $h^m(\mathcal{G})$ via the decomposition case of the recurrence.

This update pattern matches tensor primitives and is implemented in PyTorch (Paszke et al. 2019) as illustrated in Figure 1. The same tensor layout supports batching across multiple states or cost functions.

The entire hypergraph is processed in parallel, and the updates repeat until the vertex values converge to a fixed point. In practice, we detect convergence by copying the vertex values at the start of an iteration and checking whether the difference is exactly 0. If convergence is detected, the value $h^m(s)$ is transferred to the CPU and returned. However, this incurs significant CPU-GPU synchronization overhead. To mitigate this, we track the maximum number of iterations needed to reach a fixed point and defer checking until that threshold is reached. This may perform more iterations than necessary, but the additional cost is negligible in practice.

Dominance Pruning & Cost Partitioning

Because the hypergraph is constructed once and then reused, we can simplify it before running any convolution rounds. Dominance pruning removes hyperedges that compete for the same head vertex but can never win the final min-reduction. Let the value proposed by a hyperedge $e = (T, \{v\})$ in one convolution round be $u_e = \max_{x\in T} d(x) + \text{weight}(e)$. If another hyperedge with the same head always proposes a value no larger than u_e, then e can be removed without affecting any vertex update. Since this depends only on tail inclusion and hyperedge weights, it can be checked once when constructing the hypergraph.

Theorem 1. *Let $e_1 = (T_1, H)$ and $e_2 = (T_2, H)$ be two hyperedges. If $T_1 \subseteq T_2$ and weight$(e_1) \leq$ weight(e_2), then removing e_2 does not affect the computed vertex values.*

Proof. Let v_1 and v_2 be the maximum values computed from the tail vertices T_1 and T_2, respectively. Since $T_1 \subseteq T_2$, we have $v_1 \leq v_2$. Because weight$(e_1) \leq$ weight(e_2), it follows that $v_1 + \text{weight}(e_1) \leq v_2 + \text{weight}(e_2)$. The final update takes the minimum over hyperedges with head H, so e_2 can never improve on the value proposed by e_1 and therefore has no effect on the computed vertex values. □

In Theorem 1, the edges e_1 and e_2 share the same singleton head, so they compete in the same min-reduction. The tail vertices capture how expensive it is to satisfy the regressed precondition of the underlying action, and adding vertices to the tail can only increase the max-reduction.

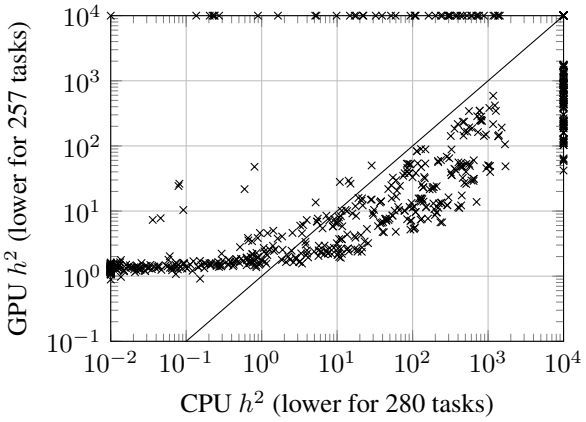

Figure 2: Per-instance comparison of the runtime of the CPU version of h^2 (x-axis) and the GPU version (y-axis).

Cost Partitioning. We adopt the cost partitioning mechanism suggested by Haslum (2006) to make further use of the parallel computation of GPUs. The idea of dominance pruning extends to the cost-partitioned setting, but the pruning condition becomes stricter. A cost partitioning consists of cost functions $c_1, \ldots, c_n$ that distribute the cost of each action a such that each $c_i(a)$ lies between 0 and cost(a) (Katz and Domshlak 2010; Seipp, Keller, and Helmert 2020). We use the notation $h^m(s \mid c)$ to indicate that h^m is computed using $c(\cdot)$ instead of cost$(\cdot)$. Additionally, the cost functions must satisfy the constraint that for each action a, $c_1(a) + \cdots + c_n(a) \leq \text{cost}(a)$. Under this constraint, the additive heuristic $h^m(s \mid c_1, \ldots, c_n) = \sum_{i=1}^n h^m(s \mid c_i) \leq h^*(s)$ remains admissible (Katz and Domshlak 2010; Seipp, Keller, and Helmert 2020). In our implementation, all cost functions share the same underlying hypergraph structure. This requires additional care when pruning hyperedges: a hyperedge can only be removed if the induced dominance condition $c_i(a_1) \leq c_i(a_2)$ holds for all cost functions c_i.

Experimental Evaluation

We evaluate our approach on the benchmarks from the optimal tracks of the International Planning Competition. Our implementation extends the Fast Downward planning system (Helmert 2006), using LibTorch 2.1.2 and CUDA version 12.1 for GPU computation. We use the Lab library (Seipp et al. 2017) to automate our evaluation, limiting runtime and CPU memory to 30 min and 8 GiB. For the GPU-based heuristics, we utilize a single NVIDIA A100 GPU with 40 GB of memory for each task and a single core of an AMD EPYC 7742 CPU with 8 GB of RAM.

Our evaluation compares a single-threaded h^2 implementation on the CPU, which implements the generalized Bellman-Ford algorithm (Haslum 2006), against four variants of h^2 on the GPU: (1) a base variant that computes h^2 using convolution operations as described above, (2) a batched version thereof that computes the heuristic for all successors of an expanded state in parallel (B$-h^2$), and two variants using cost partitioning with five different cost func-

Coverage		CPU h^2	CPU LMc	GPU h^2	GPU B$-h^2$	GPU $h^2_{\mathrm{CP}^r}$	GPU $h^2_{\mathrm{CP}^g}$
airport	50	16	**28**	15	15	14	14
barman-11	20	0	**4**	**4**	**4**	0	0
blocks	35	17	**28**	18	18	18	18
data-network	20	11	**12**	11	**12**	11	11
depot	22	2	**7**	4	4	4	4
driverlog	20	7	**14**	9	10	8	8
elevators-08	30	9	**22**	16	19	15	15
elevators-11	20	7	**18**	13	16	12	12
freecell	80	8	**15**	**15**	**15**	10	10
ged	20	13	**16**	15	15	13	13
gripper	20	6	**7**	6	**7**	6	6
hiking	20	7	10	10	**11**	8	9
logistics00	28	10	**20**	10	10	13	14
miconic	150	45	**141**	45	50	45	45
mprime	35	22	**23**	19	19	19	17
mystery	30	**17**	**17**	11	11	11	9
openstacks-08	30	12	**23**	14	15	12	12
openstacks-11	20	7	**18**	9	10	7	7
organic	20	**7**	**7**	5	5	5	3
organic-split	20	14	**17**	2	2	2	1
parcprinter-08	30	13	**19**	15	15	13	14
parcprinter-11	20	9	**14**	11	11	9	10
pegsol-08	30	26	**29**	27	27	27	27
pegsol-11	20	16	**19**	17	17	17	17
petri-net-alignment	20	10	9	**12**	**12**	9	9
pipesworld-notankage	50	12	**18**	14	14	12	12
pipesworld-tankage	50	8	**12**	8	8	7	6
psr-small	50	48	**49**	48	48	48	47
satellite	36	4	**8**	6	6	5	5
scanalyzer-08	30	6	**16**	6	9	6	6
scanalyzer-11	20	3	**13**	3	6	3	3
snake	20	3	**7**	0	0	0	0
sokoban-08	30	13	**30**	22	22	19	20
sokoban-11	20	10	**20**	18	18	16	17
spider	20	5	**11**	0	0	0	0
storage	30	12	**15**	14	14	13	13
termes	20	2	**6**	3	3	2	2
tetris-14	17	3	**6**	2	2	1	0
tidybot-11	20	8	**14**	0	0	0	0
tpp	30	5	**7**	6	6	5	6
transport-11	20	6	6	**7**	**7**	6	6
transport-14	20	4	6	6	**7**	5	6
trucks	30	5	**10**	6	8	5	5
woodworking-08	30	7	**19**	8	10	8	8
woodworking-11	20	2	**13**	3	5	3	3
others	454	103	146	103	103	103	103
Sum	1827	576	**973**	612	642	571	569

Table 1: Per-domain coverage (number of solved instances) comparing all planner configurations. Domain shading encodes parallelization potential: average $|E|$ vs. the A100's ≈221k resident threads' ratio (darker green = more saturated; gray = out of memory).

tions, (3) one based on creating one partition per goal fact (sampling at most 5 goals) and assigning actions (with the full cost) to the partition where they have the largest impact ($h^2_{\mathrm{CP}^g}$) (Haslum 2006), and (4) another using random cost partitioning ($h^2_{\mathrm{CP}^r}$). As a strong baseline built on h^1, we include LM-cut in the comparison (Helmert and Domshlak 2009). We also experimented with variants of h^1 and h^3 on the GPU, but found that the former does not benefit from parallelization, presumably because the CPU computation is too cheap, and that the latter runs into memory issues even for medium-sized instances. Thus, we focus our experiments on the critical path heuristic with $m = 2$, using A^* search for all heuristics (Hart, Nilsson, and Raphael 1968).

For the GPU-based implementation, we construct the hypergraph on the CPU as a preprocessing step and then transfer it to the GPU for heuristic computation. We then use

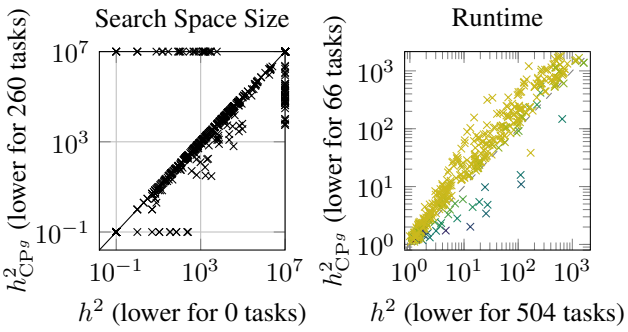

Figure 3: Per-instance comparison of search-space size (left plot) and the runtime (right plot) of the h^2 without (x-axis) and with cost partitioning (y-axis). Deeper blue hues indicate a smaller search space for $h^2_{\mathrm{CP}^g}$ compared to h^2.

Torch operations to perform the convolution steps on the GPU to leverage the parallel processing capabilities effectively. For further optimization, we utilize CUDA graphs to reduce the overhead associated with launching GPU kernels.

Figure 2 shows a per-instance runtime comparison of GPU-based B$-h^2$ with the CPU baseline. We observe a clear advantage for large instances, with speed-ups of up to two orders of magnitude. There is an initial overhead for constructing the hypergraph and initializing the GPU.

Table 1 shows the coverage of all heuristics. Domains in which all h^2 variants perform identically are summarized under "others". Our main observation is that the parallelization on the GPU leads to a strong performance improvement, with 36 additional instances solved. Enabling batching results in another 30 instances being solved. The variants using cost partitioning are not competitive in terms of coverage, as they run out of memory frequently and the increased runtime overhead is not worth it compared to the information gain. None of the parallel h^2 variants can compete with LM-cut overall, but there are positive exceptions in some domains.

Figure 3 sheds further light on the accuracy and runtime of the cost-partitioning heuristics. We show results of the goal-based cost partitioning, which tends to yield a better heuristic. The left plot shows the search-space reduction, indicating that although there is a reduction in many instances, it is not substantial for most of them. Where it does not improve the heuristic, the cost partitioning leads to a measurable runtime overhead, as can be seen in the right plot.

Conclusion

We present a GPU implementation of the critical path heuristic h^m for classical planning, formulating its dynamic program as repeated convolutions on a fixed directed hypergraph. It yields substantial speedups over the CPU baseline, up to two orders of magnitude on large instances. The resulting planner solves more instances overall, especially with batching. Although it does not yet match heuristics like LM-cut and cost partitioning remains limited by memory overhead, our results show that GPUs can be utilized in planning, with heuristic evaluation a promising entry point.

Acknowledgements

This work was funded by the Swiss National Science Foundation (SNSF) as part of the project "Unifying the Theory and Algorithms of Factored State-Space Search" (UTA). This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The computations were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS), partially funded by the Swedish Research Council through grant agreement no. 2022-06725. The research has been supported by the Alexander von Humboldt Foundation with funds from the Federal Ministry for Education and Research, Germany, by the European Research Council (ERC), Grant agreement No. 885107, and by the Excellence Strategy of the Federal Government and the NRW Länder, Germany.

References

Alcázar, V.; and Torralba, Á. 2015. A Reminder about the Importance of Computing and Exploiting Invariants in Planning. In Brafman, R.; Domshlak, C.; Haslum, P.; and Zilberstein, S., eds., *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, 2–6. AAAI Press.

Bellman, R. E. 1958. On a routing problem. *Quarterly of Applied Mathematics*, 16: 87–90.

Bonet, B.; and Geffner, H. 2001. Planning as Heuristic Search. *Artificial Intelligence*, 129(1): 5–33.

Edelkamp, S.; and Sulewski, D. 2009. Parallel State Space Search on the GPU. In *Proceedings of the Second Annual Symposium on Combinatorial Search (SoCS 2009)*.

Fikes, R. E.; and Nilsson, N. J. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2: 189–208.

Ford, L. R. 1956. *Network Flow Theory.* Santa Monica, CA: RAND Corporation.

Gallo, G.; Longo, G.; and Pallottino, S. 1993. Directed Hypergraphs and Applications. *Discrete Applied Mathematics*, 42(2): 177–201.

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.

Haslum, P. 2006. *Admissible Heuristics for Automated Planning*. Ph.D. thesis, Linköping University, Sweden.

Haslum, P.; Bonet, B.; and Geffner, H. 2005. New Admissible Heuristics for Domain-Independent Planning. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI 2005)*, 1163–1168. AAAI Press.

Haslum, P.; and Geffner, H. 2000. Admissible Heuristics for Optimal Planning. In Chien, S.; Kambhampati, S.; and Knoblock, C. A., eds., *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2000)*, 140–149. AAAI Press.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26: 191–246.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 162–169. AAAI Press.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14: 253–302.

Katz, M.; and Domshlak, C. 2010. Optimal admissible composition of abstraction heuristics. *Artificial Intelligence*, 174(12–13): 767–798.

Kipf, T. N.; and Welling, M. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the Fifth International Conference on Learning Representations (ICLR 2017)*. OpenReview.net.

Kishimoto, A.; Fukunaga, A.; and Botea, A. 2013. Evaluation of a simple, scalable, parallel best-first search strategy. *Artificial Intelligence*, 195: 222–248.

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; Desmaison, A.; Kopf, A.; Yang, E.; DeVito, Z.; Raison, M.; Tejani, A.; Chilamkurthy, S.; Steiner, B.; Fang, L.; Bai, J.; and Chintala, S. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the Thirty-third Annual Conference on Neural Information Processing Systems (NeurIPS 2019)*, 8024–8035.

Seipp, J.; Keller, T.; and Helmert, M. 2020. Saturated Cost Partitioning for Optimal Classical Planning. *Journal of Artificial Intelligence Research*, 67: 129–167.

Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. https://doi.org/10.5281/zenodo.790461.

Shipovalov, E.; and Pryanichnikov, V. 2020. Scalable State Space Search on the GPU with Multi-Level Parallelism. In *Proceedings of the 19th International Symposium on Parallel and Distributed Computing (ISPDC 2020)*, 84–92.

Steinmetz, M.; and Torralba, Á. 2019. Bridging the Gap between Abstractions and Critical-Path Heuristics via Hypergraphs. In Lipovetzky, N.; Onaindia, E.; and Smith, D. E., eds., *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*, 473–481. AAAI Press.

Sulewski, D.; Edelkamp, S.; and Kissmann, P. 2011. Exploiting the Computational Power of the Graphics Card: Optimal State Space Planning on the GPU. In Bacchus, F.; Domshlak, C.; Edelkamp, S.; and Helmert, M., eds., *Proceedings of the Twenty-First International Conference on Automated Planning and Scheduling (ICAPS 2011)*, 242–249. AAAI Press.

Taitler, A.; Alford, R.; Espasa, J.; Behnke, G.; Fišer, D.; Gimelfarb, M.; Pommerening, F.; Sanner, S.; Scala, E.; Schreiber, D.; Segovia-Aguas, J.; and Seipp, J. 2024. The 2023 International Planning Competition. *AI Magazine*, 45(2): 280–296.