Parallel Search For Map-Based Pathfinding

Jason Gedge
Department of Computing Science
University of Alberta
gedge@cs.ualberta.ca

Abstract

With the advent of multicore processors, parallelization of searching algorithms becomes an area of focus. To maximize the gain when applying parallelization we must find ways of creating synchronization-free algorithms with low amounts of idle time for all running processes. We implement an existing algorithm with minimum synchronization time: parallel structured duplicate detection (PSDD). We show that a large amount of gain can be obtained when applying PSDD to the sliding-tile puzzle domain, but see a loss in performance when applied to map-based pathfinding. We analyze this issue and outline the reasons for why we see such a performance loss. Finally, we present alternative parallel solutions for map-based pathfinding.

Introduction

Single-agent search is a fundamental technique used in many areas of AI. The major problem is that it is oftentimes both computationally and memory-intensive. Particularly, computational expense is more of a concern than memory since vast amounts of memory are available in modern computers. Although processor speed has increased significantly since the advent of the PC, we are finally seeing a limit to the amount of speed we can achieve using a single core processor. This has been the guiding motivation behind multicore processors. With two and four-core processors being regular commodities, one might ask the question of how can we take advantage of such computing power?

The simple answer to this question is to focus our research into search algorithms which are parallel-friendly. Gustafson's law tells us that for P processors and α, the proportion of our code that is sequential, we can achieve a speedup factor of $P - \alpha \cdot (P - 1)$. The key to maximizing our speedup then is to minimize the amount of sequential code, assuming a fixed number of processors. The first way to so is to minimize the amount of synchronization required between multiple processes[1], since this can only be done sequentially. The second is to reduce the amount of time any one process idles due to synchronization, insufficient work available, or for any other reason. The idling of a process is effectively reducing the value of P. These two things play a key role in creating high-performance, parallel-friendly algorithms. Minimal idling time can be achieved by an effective means of distributing work evenly across all processes.

This project has approached the problem by first implementing an existing algorithm, called *parallel structured duplicate detection* (PSDD), and analyzing it on various domains. PSDD is a simple extension to SDD to allow for parallelization, both of which rely on locality-preserving projections and layer-by-layer searching technique. It was found that various characteristics of a domain, and the heuristic used to guide the search, contribute towards performance gains when using PSDD. To optimize performance it was found that there has to be a fine balance between how often the f-cost boundary must be expanded and the amount of work to be done for each expansion. For example, with map-based pathfinding and an octile distance heuristic there is often a large number of boundary expansions and a low amount of work at each expansion, both of which contribute to a performance loss.

Background

Parallelism can be applied in many different situations: from shared-memory to distributed-memory models, SIMD to MIMD architectures, and local to distributed contexts. The focus area of this paper is on a shared-memory, multicore model aimed at the consumer level.

One of the more naive approaches to parallelism was given by Ferguson and Korf in (Ferguson and Korf, 1988). The idea is to simply assign a processor

[1] Note that the term process used in this report does not necessarily mean an actual operating system process, but any processing unit involved in an algorithm. This could be an actual operating system process, thread, or some other mechanism to achieve parallelism.

a number of processes and let it allocate them as it pleases. Once all processors have been been depleted by a process, it spawns a copy of itself with a new pool of processors and then blocks, awaiting an answer before it continues. The simplicity and generality of this approach comes at the cost of not being fine-tuned for any specific properties of a domain. The authors report a speedup of only 12 on a 32-node system (Ferguson and Korf, 1988).

More recent algorithms have approached parallel search in a variety of ways. Both (Korf and Schultze, 2005) and (Zhang and Hansen, 2006) approach the problem of a large-scale breadth-first search, in which the complete state space will be searched. In this case there is no gain in using heuristic-based searching techniques, such as A* (Hart et al., 1968) or IDA* (Korf, 1985), since we have no goal. These algorithms often use external memory to store state space information, so parallelization shows significant improvement, sometimes by a factor greater than the number of processors. This is mainly due to the IO-bound nature of similar sequential implementations.

Parallel structured duplicate detection can be guided by any search algorithm that expands a layer at a time, and hence can be developed to search a complete state-space or for more goal-oriented searches, depending on which searching technique is used (Zhou and Hansen, 2007). SDD projects the main state space into an abstract space and uses this abstract space to partition generated and expanded nodes into smaller sets (Zhou and Hansen, 2004). Since the projection function preserves locality, we are able to determine neighbouring abstract nodes, which informs us which information is required to be loaded into internal memory. This concept is referred to as a *duplicate-detection scope*. PSDD expands on SDD by noticing that disjoint duplicate-detection scopes allow multiple processes to work in parallel without the requirement for synchronization.

Structured Duplicate Detection

We begin with a detailed look into structured duplicate detection (SDD), since it is the backbone of PSDD. SDD is one of many approaches to graph search in which local structure of the graph is exploited so that duplicate detection can be performed immediately, instead of begin delayed like other approaches (Edelkamp et al., 2004; Korf, 2004).

Given a state-space projection function p and a state-space graph G we can form the abstract state-space graph $G' = (V', E')$ as follows:

$$\begin{aligned} V' &= \text{image}(p) \\ E' &= \left\{(x', y') \mid \exists\,(x, y) \in E \text{ s.t. } p(x) = x', p(y) = y'\right\}. \end{aligned}$$

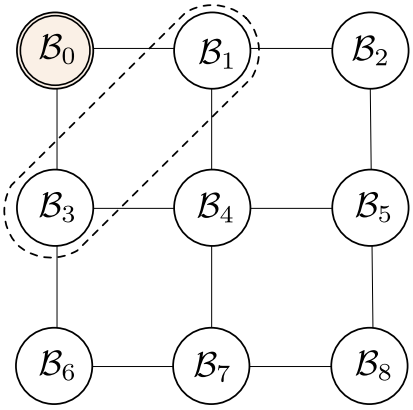

Figure 2: An example of an abstract state-space graph for the 8-puzzle in which we project states based on the location of the blank. The abstract state $\mathcal{B}_i$ corresponds to the blank being at position i in the original state-space. The circled nodes are those in the duplicate-detection scope of any node x such that $p(x) = \mathcal{B}_0$.

An example of a projection function for the 8-puzzle would be to simply ignore all but the position of the blank. This results in the abstract state-space graph seen in Figure 2. In general, we can usually create a projection function by discarding state information.

All states in the original state-space who project to the same abstract state form a set which we call an $n$block. This partitioning of states is what allows detection to be performed immediately. Since our projection function preserves locality, we know from which $n$blocks generated nodes will come from through successor relationships in the abstract state-space graph. This forms the fundamental idea of a *duplicate-detection scope*. The duplicate-detection scope of a state x is defined as:

$$\text{DD}(x) = \{\ y \mid p(y) \in \text{successors}(p(x))\ \}.$$ [2]

SDD is intended to be used alongside a search algorithm which expands a layer of nodes at a time, such as breadth-first search. This allows us to choose a single $n$block for expansion and require us to have only those $n$blocks in its duplicate-detection scope located in internal memory. Other $n$blocks can be stored on external memory and paged into internal memory when needed.

[2] Note that this definition is slightly different than the definition given in (Zhou and Hansen, 2007).

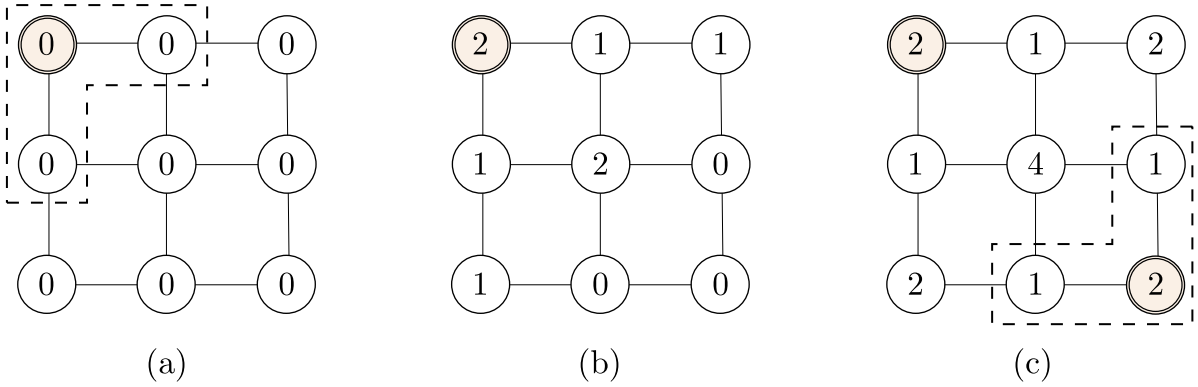

Figure 1: A series of σ-table updates. (a) $\mathcal{B}_0$ is selected for processing and attains a lock on those nodes in its duplicate-detection scope. (b) The σ values of the abstract nodes after $\mathcal{B}_0$ is selected for processing. (c) The resulting σ values after $\mathcal{B}_8$ is chosen for processing, which is possible since it had a $sigma$ value of 0. It is unnecessary to update the σ value of a node selected for processing since it will always be 0 before and after processing.

Parallel Structured Duplicate Detection

Parallel structured duplicate detection (PSDD) is a very straightforward extension to SDD that allows for low amounts of synchronization required between processes. It exploits the local structure of the search graph to find $n$blocks which can be processed without worrying about another process generating the same states.

The key concept is that of *disjoint* duplicate-detection scopes. Two states x, y in the original state-space graph are said to have disjoint duplicate-detection scopes iff $\mathrm{DD}(x) \cap \mathrm{DD}(y) = \emptyset$. A straightforward theorem based on this definition is that x and y cannot share a common successor node if their duplicate-detection scopes are disjoint (Zhou and Hansen, 2007). We can take advantage of this idea by allowing multiple processes to work on $n$blocks whose nodes have disjoint duplicate-detection scopes. Since there are no common successors, each processor can be given exclusive rights to neighbouring $n$blocks.

Given this observation, it is then required to find a means of detecting disjoint duplicate-detection scopes. PSDD proposes the concept of a σ-table, where each abstract state is associated with a counter value that indicates how many successor states are being used by other processes. From this definition it follows that a σ value of 0 corresponds to a state which is available for processing. When an abstract state y is chosen for expansion we increment the σ values of (1) the successors of y and (2) the successors of y's successors. Whenever processing of an $n$block is complete we then apply the same process but decrement the σ values instead of incrementing. The update of the σ-table is, in essence, the only time a mutex lock is required.

```
    input  : States start, goal and projection function p(·)
    output: The path from start to goal

 1  Initialize nblocks
 2  limit ← h(start)
 3  while goal not found do
 4      newlimit ← ∞
 5      while ∃ states with f-cost ≤ limit do
 6          β ← next available nblock
 7          foreach state s ∈ OPEN(β) do
 8              closed(s) ← true
 9              if s = goal then break
10              foreach successor c_i of s do
11                  if closed(c_i) then continue
12                  if f(c_i) ≤ limit then
13                      append c_i to nblock p(c_i)
14                  else
15                      newlimit ← min{newlimit, f(c_i)}
16      limit ← newlimit
17  reconstruct path
18  return path
```

Algorithm 1: $SDD(start, goal, p)$, using an IDA*-like strategy for layer-by-layer expansion.

Since this is a relatively inexpensive operation, it does not have a significant impact on performance. A series of σ-table updates can A be seen in Figure 1. Considering the SDD algorithm given in Algorithm 1, PSDD is mostly just a modification of line 6 to fetch an available $n$block and perform sigma table updates.

Since the granularity of the abstract state-space graph is completely dependent on the chosen projection function, we can simply modify the projection function so as to maximize the potential for processes to work in parallel. It is important to note that there is a fine balance that must be met. One might initially think that we can make a very fine projection function to ensure that all processes are working in parallel instead of idling, but a projection

function that is too fine will have smaller amounts of work to be processed at each layer but an increased amount of synchronization. Although PSDD has an inexpensive synchronization operation, if it occurs too often we lose that bonus.

Map-based Pathfinding

A very common domain for searching, especially in video games, would be that of maps. The goal of map-based pathfinding is to find a path from one point to another point in a world that contains obstructions (e.g., walls, impassable terrain features, houses, etc.). In many contexts, such as pathfinding for video games, we can introduce low amounts of suboptimality without worry. Nevertheless, this project is concerned with finding optimal paths. In contrast to the sliding-tile and many other puzzle domains, the state-space for maps are of a polynomial size ($O(N^d)$ for d-dimensional searching, N nodes in the map). Although this initially sounds like a great thing, it does not bode well for PSDD unfortunately, as we will show in later this section.

Most previous papers that specifically target map-based pathfinding do not target parallelism, but rather find other means of improving search. Partial-Refinement A* (Sturtevant and Buro, 2005) obtains a rough initial guess of the path through an abstraction, and further refines the path later, possibly through multiple levels of abstraction. This fits well for situations when an agent should act upon a movement request immediately, such as video games. Various optimizations can be used to improve the running time, such as restricting search to a corridor at each refinement level. Hierarchical Pathfinding A* (Botea et al., 2004) also uses abstractions to find suboptimal paths, but rather abstracts to a single level and finds optimal paths between a set of entry/exit points defined along the borders of abstract cells. More recently, true distance heuristics (Sturtevant et al., 2009) were introduced to sacrifice memory for speed by storing several single-source shortest path matrices and using them as a heuristic. Since this only affects heuristic values, any search guided by a heuristic could use this technique. Botea et al. (2004) give a more in-depth review of other pathfinding techniques previously developed.

If we consider a simple octile heuristic, several properties of the map domain make it unsuitable for a parallel approach like PSDD. First, states are not uniformly distributed across all abstract states under simple map abstractions (e.g., grid abstraction, polar abstraction). In contrast, the sliding-tile puzzle is very much the opposite, where we see that the distribution of states across the abstraction is much more uniform. This presents a problem when an $n$block is chosen for expansion, since neighbouring $n$blocks will be locked down and unable to be processed. In a map domain with an octile heuristic, each f-cost boundary is relatively small and located within a relatively small region so we find a lot of processes become idle due to insufficient work available (see Figure 3). To overcome this we can increase the granularity of our abstraction, but then we find that each $n$block has so little work that the cost of locking it down and releasing it is no longer amortized over the cost of processing each $n$block.

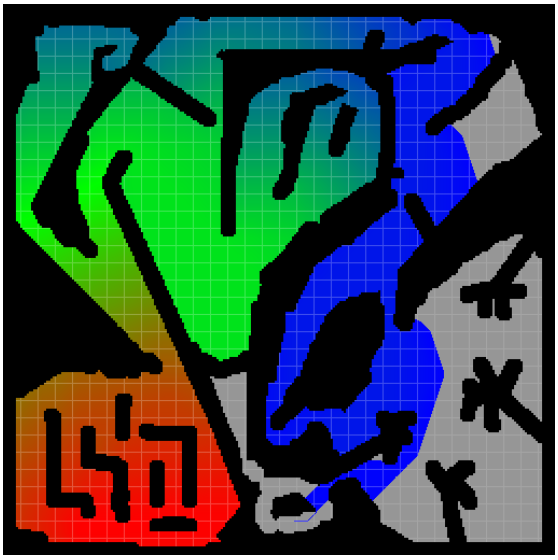

Figure 3: An example of a map (512×512) and an overlaying grid abstraction (16×16). The colours indicate various f-costs of nodes as an A* search progresses (octile heuristic). Note the high number of unique f-costs and the relatively small amount of nodes having the same f-cost.

Second, considering the fact that each f-cost layer has a relatively low amount of work, the cost of guiding the search to the next layer starts to become significant. As can be seen in the results, PSDD is more costly than even a generalized A* implementation. This is more apparent in our own implementation in which a separate thread is responsible for moving the search from one layer to the next.

PSDD becomes even more undesirable when we create specialized A* implementations specifically for map-based pathfinding. One observation in a map with octile movements is that the range of possible f-costs on the OPEN list at any time is 6[3]. This is because when we expand a node with an f-cost of α, in the worst case we move diagonally, which costs 3, and our octile heuristic value increases by

[3]This assumes that a cardinal movement is of cost 2 and a diagonal movement is of cost 3.

```
   input  : States start, goal
   output: The path from start to goal

1  Initialize seven buckets
2  currentBucket ← 0
3  costOfCurrentBucket ← f(start)
4  while goal not found and states to process do
       // Process current bucket
5      foreach state s ∈ buckets[currentBucket] do
6          closed[s_x][s_y] ← true
7          if s = goal then break
8          foreach successor c of s do
9              if closed[c_x][c_y] then continue
10             index ← f(c) − costOfCurrentBucket
11             append c to buckets[index]

       // Find next bucket
12     for i = 0, 1, ..., 6 do
13         index ← (currentBucket + i)  mod 7
14         if buckets[index] is not empty then
15             empty buckets[currentBucket]
16             currentBucket ← index
17             costOfCurrentBucket + +
18             break

19     if no bucket found then break
20 reconstruct path
21 return path
```

Algorithm 2: $Map - A^*(start, goal)$

its maximum, which is also 3. When implementing A*, we can create a set of buckets for the OPEN list, with nodes of the same f-cost being placed in the same bucket. If we ignore tie-breaking then we simply choose the bucket of lowest f-cost and expand all nodes in that bucket. A successor of an expanded node is then placed in the bucket corresponding to its f-cost. A circular structure, such as a ring buffer, is best for storing these buckets. Also, various data storage structures used by A* can be implemented as a two-dimensional array instead of a general has table, such as the CLOSED list. In our own implementation, which we refer to as Map-A*, we achieved a speedup factor of approximately 5-8 over regular A* when introducing these optimizations. Note that similar optimizations could be used in other domains where similar behaviour occurs.

Two approaches to parallelize Map-A* resulted in running times that were 5-10 milliseconds slower than Map-A* on average. These approaches include:

1. Assigning each thread a set of buckets to which it could generate nodes. All threads would process the entire current bucket and generate only those nodes that would be placed in buckets assigned to the thread. The good thing with this approach was that no synchronization was required, but the high level of redundancy in node generation actually ended up making this approach slower than Map-A*.
2. If the current bucket had a sufficient number of nodes, divide the work amongst multiple threads. Each thread would maintain its own set of buckets for generated nodes but would have to obtain a lock to merge these nodes into the primary list of buckets for future processing. The benefit to this approach was that redundant processing of the current bucket was eliminated, but the merging process prevented performance gains over Map-A*.

Results

Results for the sliding-tile puzzle were obtained from a PC with an Intel Quad Core (2.4 GHz) and 4GB of RAM. As one can see in Table 1, there is a significant gain when using PSDD but we do see a drop in the speedup factor as we increase the number of threads, as is often seen in parallel algorithms.

Method	Depth vs. Time (s)			
	40	46	48	50
A*	0.06	9.79	2.06	405.26
IDA*	0.21	4.47	0.76	59.14
PSDD (1 thread)	0.22	3.23	2.50	19.85
PSDD (2 threads)	0.17	1.92	1.87	12.13
PSDD (3 threads)	0.18	1.41	1.68	9.19
PSDD (4 threads)	0.18	1.28	1.72	7.92

Table 1: Results for various problems on the 15-puzzle. A 2-tile abstraction was used with PSDD to allow for full advantage of 4 threads. An abstraction of just the blank has many cases where a thread will idle (when using 4 threads).

Map-based pathfinding results were obtained on a laptop with an Intel Core 2 Duo (2.4 GHz) with 2GB of RAM. All times include the cost to reserve memory for all structures used in the search. As indicated previously, PSDD performs poorly compared to an A* implementation geared towards map-based pathfinding. In Figure 4 we can see how PSDD actually suffers compared to other algorithms on a set of 100 random problems for the map in Figure 3. Table 2 shows the average of 100 runs on a single problem in the same map, with a problem at depth 1428.5. Octile heuristics were used in all tests cases and a 64×64 rectangular grid abstraction was used for PSDD. A certain unfairness may be present in the results when one considers the fact that our PSDD implementation was written to be general, but considering it is still worse than a generalized A* implementation, fine-tuning it specifically for maps will probably show similar behaviour when it is compared to Map-A*.

Table 2 shows exactly how the granularity of the abstraction has to be chosen wisely based on the

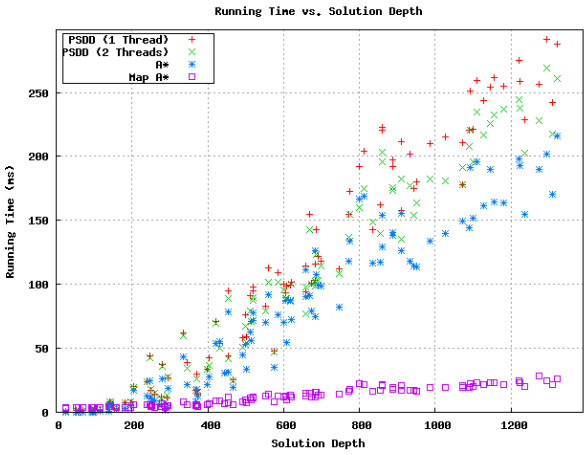

Figure 4: A scatter plot showing the running times of various algorithms versus the solution depth. Map-A*, an A* implementation written with map-based pathfinding in mind, shows significantly better running times over other algorithms.

domain. If one chooses a grid size that is too small, then threads will rarely be idle, but eventually the task of locking $n$blocks and updating the σ-table becomes more work than the processing of an $n$block. If one chooses a grid size too large, then no parallelism can be achieved since choosing an $n$block for expansion locks most other $n$blocks . Even if cells are left unlocked, the locality of open nodes in a map generally leads to a single thread processing at one time. In other words, large abstractions often degenerate PSDD into a "sequential" algorithm.

To further show that the issues outlined above are attributing to the slow running times, a differential heuristic was introduced (see Figure 5), which distributes the work better in each layer and also reduces the number of layer expansions that need to be performed to reach the goal. The running times in Table 3 show the results of this modification. What we see is that more speedup is obtained in PSDD than in A* due to the addition of a differential heuristic. This is primarily due to the number of boundary expansions and the distribution of work introduced by a differential heuristic, which benefits PSDD more than A*.

Conclusions and Future Work

Parallelization proves to be a strong method of high-speed searching in exponential domains, such as sliding-tile puzzles. If enough internal memory is available for such problems, PSDD offers speedup factor of up to 50 times over A* and 7 times over IDA* when using 4 threads. For problems where external memory must be used, this factor will most likely drop some, but this was not tested. On the other hand, map-based pathfinding appears to be a domain that is impervious to parallelization, or is it?

	Time (ms)
PSDD, 512×512 *abstraction*	
1 thread	278.57
2 threads	285.65
PSDD, 128×128 *abstraction*	
1 thread	302.71
2 threads	286.38
PSDD, 32×32 *abstraction*	
1 thread	348.66
2 threads	317.41
PSDD, 8×8 *abstraction*	
1 thread	1038.75
2 threads	1435.49
Other	
A*	202.88
Map-A*	24.47

Table 2: Results for a problem with a solution of depth 1428.5 for the map in Figure 3, averaged over 100 runs. Note that on the same machine as was used for the 15-puzzle, Map-A* runs closer to 18-19 ms on average.

Method	**Time (ms)**	**Speedup**
A*	100.02	2.03
PSDD (1 thread)	108.15	3.22
PSDD (2 threads)	108.48	2.93

Table 3: Results for the same testing scenario as in Table 2 but with the addition of a 2-node differential heuristic. A 32×32 abstraction was used for PSDD.

It may be the case that the various approaches used in this paper fail to offer improvements over sequential algorithms, but there may still be an effective means of performing map-based pathfinding in parallel. Nevertheless, considering the fact that a high-speed sequential implementation is possible, namely Map-A*, it appears that creating a map-based pathfinding algorithm that is both optimal and parallel may be a difficult task. On one hand, is relatively easy to create a synchronization-free algorithm that requires redundant processing. On the other hand, redundant processing can easily be eliminated if we allow synchronization.

Future work would consist of trying to find an efficient parallel solution to map-based pathfinding. A short-term goal is to reduce the running time to 10 milliseconds on average for the problem used to obtain the results of Table 2. To achieve such a goal, emphasis will have to be placed on parallelism that is both idle-free and synchronization-free. It may be interesting to consider looking into bidirectional heuristic search (Pohl, 1969), or some variation on this. For example, while one process

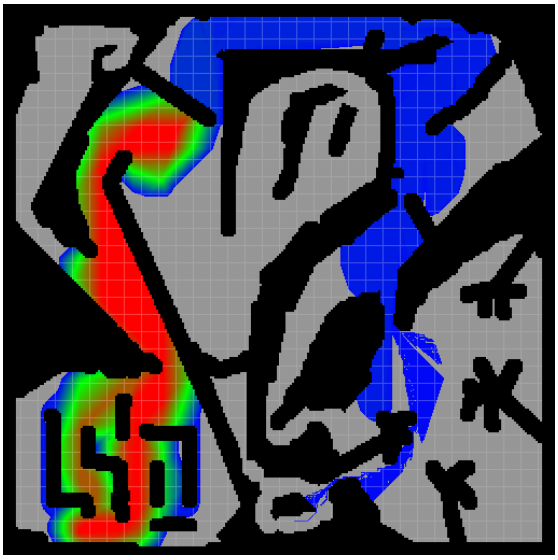

Figure 5: The same map and problem as in Figure 3 but using a differential heuristic instead of an octile heuristic.

moves the search forward, another process proceeds backwards from the goal until the two processes meet. The backward process would store optimal path lengths which could then be used as a perfect heuristic to guide the forward search the rest of the way. Another possibility is to relax the optimality constraint and see if there may be an effective means of parallelism that introduces small amounts of suboptimality.

References

Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical path-finding. *Journal of Game Development*, 2004.

Stefan Edelkamp, Shahid Jabbar, Stefan SchrÃűdl, and Baroper Str. External a*. In *In German Conference on Artificial Intelligence (KI)*, pages 226–240. Springer, 2004.

C. Ferguson and R. E. Korf. Distributed tree search and its application to alpha-beta pruning. In *Proceedings of 7th National Conference on Artificial Intelligence*, pages 128–132, 1988.

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*, SSC-4(2):100–107, 1968.

R. Korf. Best-first frontier search with delayed duplicate detection. In *In 19th National Conference on Artificial Intelligence (AAAI'04)*, pages 650–657, 2004.

R. E. Korf and P. Schultze. Large-scale parallel breadth first search. In *Proceedings of 20th National Conference on Artificial Intelligence*, 2005.

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

Ira Sheldon Pohl. *Bi-directional and heuristic search in path problems*. PhD thesis, Stanford, CA, USA, 1969.

Nathan Sturtevant, Ariel Felner, Max Barer, Jonathan Schaeffer, and Neil Burch. Memory-based heuristics for explicit state spaces. In *Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09)*, 2009.

Nathan R. Sturtevant and Michael Buro. Partial pathfinding using map abstraction and refinement. In *In 20th National Conference on Artificial Intelligence (AAAI'05)*, pages 1392–1397, 2005.

Y Zhang and E. Hansen. Parallel breadth-first heuristic search on a shared-memory architecture. In *Workshop on Heuristic Search, Memory-Based Heuristics and Their Applications*. AAAI, July 2006.

Rong Zhou and Eric A. Hansen. Parallel structured duplicate detection. In *AAAI*, pages 1217–, 2007. URL http://www.informatik.uni-trier.de/~ley/db/conf/aaai/aaai2007.html#ZhouH07.

Rong Zhou and Eric A. Hansen. Structured duplicate detection in external-memory graph search. In *AAAI*, pages 683–689, 2004. URL http://www.informatik.uni-trier.de/~ley/db/conf/aaai/aaai2004.html#ZhouH04a.