Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Martin Milichovský

Study programme: Open Informatics

Specialisation: Computer and Information Science

Title of Bachelor Project: Parallel Trajectory Planning

Guidelines:

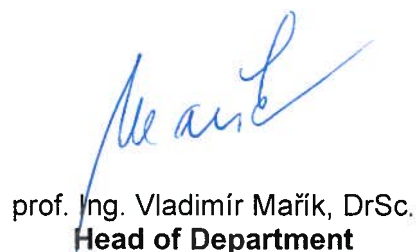
1. Study the recommended literature, focus especially on the ways of computation paralelization on CPU.
2. Design the paralelization of the planning process using the A* and possibly AA* algorithms.
3. Implement the designed approach using the multi-agent platform Aglobe.
4. Evaluate the quality of the proposed paralelization, the quality of the plan and the speed of the planning process.

Bibliography/Sources:

[1] E. Burns, S. Lemons, R. Zhou, and W. Ruml: Best-first heuristic search for multi-core machines. In Proceedings of the 21st international jont conference on Artifical intelligence, pp. 449– 455, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.
[2] E. Burns, S. Lemons, R. Zhou, and W. Ruml. Parallel best-first search: The role of abstraction. In Proceedings of the AAAI-10 Workshop on Abstraction, Reformulation, and Approximation, 2010.
[3] M. Evett, A. Mahanti, D. Nau, J. Hendler, and J. Hendler: PRA*: Massively parallel heuristic search. Journal of Parallel and Distributed Computing, 25:133–143, 1995.
[4] A. Kishimoto, A. Fukunaga, and A. Botea: Scalable, Parallel Best-First Search for Optimal Sequential Planning. In Proceedings of the International Conference on Automated Scheduling and Planning ICAPS-09, pages 201–208, Thessaloniki, Greece, 2009.

Bachelor Project Supervisor: Bc. Štěpán Kopřiva, MSc.

Valid until: the end of the winter semester of academic year 2012/2013



prof. Ing. Vladimír Mařík, DrSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 9, 2012

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Martin Milichovský

Studijní program: Otevřená informatika (bakalářský)

Obor: Informatika a počítačové vědy

Název tématu: Paralelní plánování trajektorie

Pokyny pro vypracování:

1. Prostudujte předloženou literaturu a nastudujte možnosti paralelizace výpočtu na CPU.
2. Navrhněte způsob paralelizace plánování s použitím algoritmu A*, případně AA*.
3. Navržený způsob naimplementujte za použití multiagentní platformy Aglobe.
4. Experimentálně na benchmarkových scénářích ověřte kvalitu navržené paralelizace, tedy kvalitu nalezeného plánu a rychlost nalezení takového plánu.

Seznam odborné literatury:

[1] E. Burns, S. Lemons, R. Zhou, and W. Ruml: Best-first heuristic search for multi-core machines. In Proceedings of the 21st international jont conference on Artifical intelligence, pp. 449– 455, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.
[2] E. Burns, S. Lemons, R. Zhou, and W. Ruml. Parallel best-first search: The role of abstraction. In Proceedings of the AAAI-10 Workshop on Abstraction, Reformulation, and Approximation, 2010.
[3] M. Evett, A. Mahanti, D. Nau, J. Hendler, and J. Hendler: PRA*: Massively parallel heuristic search. Journal of Parallel and Distributed Computing, 25:133–143, 1995.
[4] A. Kishimoto, A. Fukunaga, and A. Botea: Scalable, Parallel Best-First Search for Optimal Sequential Planning. In Proceedings of the International Conference on Automated Scheduling and Planning ICAPS-09, pages 201–208, Thessaloniki, Greece, 2009.

Vedoucí bakalářské práce: Bc. Štěpán Kopřiva, MSc.

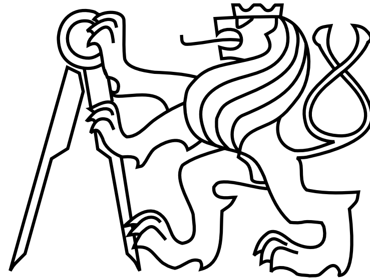
Platnost zadání: do konce zimního semestru 2012/2013

prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 9. 1. 2012

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Bachelor Project

Parallel Trajectory Planning

Martin Milichovský

Supervisor: Štěpán Kopřiva Bc., MSc.

Study Programme: Open Informatics

Field of Study: Computer and Information Science

May 15, 2012

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague on May 15, 2012 ....Martin Milichovský........

Abstract

The goal of this project is to propose and implement parallel path planning algorithm reflecting state-of-the-art methods.

The motivation is utilization of modern multicore computers for this task, commonly solved by a serial approach. This work describes main ideas and methods of the parallelization, where each thread attempts to expand most promising nodes.

There was implemented a generic framework in Java™ for parallel planning/space exploration, deploying two parallel search concepts: hash-distributed (HDA*), and work-stealing (PA*).

A serial planner for plane's trajectory has been extended to run in this framework.

The empirical results have shown that the original task can run faster in parallel. Properties and issues of each approach were discussed.

Abstrakt

Cílem projektu je navrhnout a implementovat algoritmus pro paralelní plánování na základě moderních metod.

Motivací je využití vícejádrových počítačů ke zpracování úloh, které jsou dnes obvykle řešeny seriově. Tato práce popisuje hlavní myšlenky a metody paralelizace, která spočívá v prohledávání nejslibnějších uzlů ve více vláknech.

Byl vytvořen obecný framework v jazyce Java™ pro paralelní plánování, resp. prohledávání prostoru, fungujucící podle dvou možných principů: rozdělení prostoru hašovací funkcí (HDA*), a soupeření o uzly (PA*).

Na základě těchto myšlenek byl rozšířen původní seriový plánovač trasy letadla tak, aby pracoval paralelně.

Výsledky byly experimentálně porovnány a ukázaly, že úloha řešená paralelním plánováním pracuje rychleji než původní seriová. Dále byly diskutovány vlastnosti a problematika jednotlivých přístupů.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Objectives 1
1.3 Overview 1

2 Background **3**
2.1 Path Planning 3
2.2 BFS 3
2.2.1 A* 4
2.3 AgentFly 4
2.3.1 AgentFly path planner 4
2.4 Parallelism 4
2.4.1 Overhead 4

3 State of the Art **5**
3.1 PA* 5
3.2 HDA* 5
3.3 PBNF 5

4 Analysis **7**
4.1 HDA* 7
4.2 PA* 8
4.3 Master-Worker 9
4.4 Optimality 9
4.5 Interface 10

5 Implementation **11**
5.1 Best First Search 11
5.1.1 Detachment from the state's properties 12
5.1.2 ClosedSet, OpenSet 12
5.1.3 Configuration 13
5.1.3.1 Checking closed set 13
5.1.4 Search interrupt - Timeout 14
5.1.5 Listeners 14
5.1.6 Statistics 15
5.2 AgentFly Path Planner 15

5.3 Parallel Implementation . . . 15
5.3.1 Implementation with regard to BFS . . . 16
5.3.2 Master - Worker lifecycle . . . 16
5.3.2.1 Initialization . . . 16
5.3.2.2 Master cloning . . . 17
5.3.3 Configuration . . . 18
5.3.3.1 Sharing closed set . . . 18
5.3.3.2 Sharing open set . . . 18
5.3.3.3 Sequential distribution . . . 19
5.3.4 ClosedSet, OpenSet . . . 19
5.3.5 Synchronization . . . 19
5.3.6 Exception handling . . . 21
5.4 Configuration Drilldown . . . 22
5.4.1 HDA* . . . 22
5.4.2 HDA* with shared closed set . . . 22
5.4.2.1 Sequential distribution . . . 22
5.4.3 PA* . . . 23
5.4.3.1 Suboptimal PA* . . . 23

6 Experimental Results 25
6.1 AgentFly Path Planner . . . 25
6.2 Sliding Tile Puzzle . . . 27
6.3 Search Time and Volume . . . 27
6.4 Speedup Over Serial A* . . . 27

7 Discussion 31
7.1 Observed Issues . . . 31
7.1.1 Increase of search overhead . . . 31
7.1.2 Hashing in continuous space . . . 31
7.1.3 Increase of communication overhead . . . 32
7.1.4 Increase of synchronization overhead . . . 32
7.2 AgentFly Path Planner . . . 33
7.3 15-Puzzle . . . 33

8 Conclusion 35
8.1 Future Work . . . 35

A CD Contents 39

List of Figures

4.1 Manipulation with generated nodes in HDA* 8
4.2 Manipulation with generated nodes in A* 9

5.1 Operation of a worker 21

6.1 Planner setup test 4 26
6.2 Mountain 26
6.3 The 15-Puzzle task 27
6.4 Results 28
6.5 Speedup over serial A* 29

7.1 A* timeline 32
7.2 HDA* timeline 32

Chapter 1

Introduction

1.1 Motivation

Modern computers do not increase its computing power by speeding up clock rate anymore but by adding more cores, allowing programs to run in more threads. That challenges to run complex tasks parallelly in many threads and thus requires fundamental algorithms to consider this.

State space exploration tasks require vast CPU and memory resources and the runtime directly depends on these (as in any dynamic programming tasks). Therefore planning is convenient candidate to exploit parallelism of multi-core machines.

1.2 Objectives

The aim is to propose and implement parallel extension of A* algorithm, based on the state of the art methods, for flight trajectory path planning. The idea is that more threads can explore more states in the same time. It is required to distribute the states among them in a way that *a*) doesn't have much overhead; *b*) and makes the search end faster. This is met by Hash Distributed A* (HDA*) that divides the search space by a quick hash function. A suitable solution, concerning discussed aspects, is to be proposed and implemented.

The objective is to show that the implementation makes use of the additional threads provided, and reduces the time needed for the execution.

1.3 Overview

In this work there is described the background of the solved task, its properties and issues (Chapter 2). Further there are introduced main principles of state-of-the-art methods for parallelisation (3). Next, there is chosen and analysed approach for implementation, and basic framework is defined with its fundamental functionality (4).

The implementation is went through, including low-level details of critical spots (5). The implementation is abstract and it is explained how to interpret any search task to accommodate given framework for a specific usage.

Finally it is described how the path planning is implemented. There was also a 15-Puzzle search task implemented to compare the nature of these domains.

The experimental results introduce testing tasks and their properties (6). Methodology for collecting statistically relevant results is described, and collected data are visualized in graphs, to be easily compared across approaches and search domains.

Observed relationships are then discussed and explained according to the nature of solved tasks (7). The search instances are compared for its suitability for parallelisation.

Final chapter contains concluding remarks and proposes topics of further work (8).

Chapter 2

Background

2.1 Path Planning

Generally path planning is exploring a state space that can be described as a graph of adjacent nodes (states). Edges of the graph represent basic agent's movements. When the goal in the graph is reached these edges form the desired complex path.

For search algorithms we observe following features:

- **Completeness** That means whether the algorithm finds a solution where there is any.
- **Optimality** That the path found has the lowest path cost among all solutions. [9]

To avoid infinite loop it is necessary to check for duplicate states already visited and expanded (unless the search space is a tree – has no cycles). That is a complication in a continuous search space where it is possible to fit into any ϵ-neighborhood ($\epsilon > 0$) of a state infinite number of another states.

Number of generated states grows exponentially because each state has as many successors (expansions) as is the number of available actions. Even though modern computers have enough memory to store huge state spaces but the main problem is to quickly find solution in it because the generation of states takes considerable time. And further time it takes to organise them.

Especially in continuous search space it is necessary to involve various *tricks* to reduce memory and time requirements for the search to end and still remain as complete and optimal as possible. The complication is that continuous space provides an infinite diversity of actions (e.g. step in any direction and of any length) and infinite number of possible states.

2.2 BFS

Best-first search (BFS) is a graph exploring algorithm that maintains two sets, *open* and *closed*. It chooses to explore a state $\mathbf{S}$ from the *open set* that has lowest score according to ordering function $f(\mathbf{S})$, $\mathbf{S}$ is moved into the *closed set*. If $\mathbf{S}$ is not goal its successors are generated and inserted into the *open set* unless contained in the *closed set* already.

2.2.1 A*

A* is a BFS where $f = g + h$ (path cost + heuristics estimate). A* is a popular algorithm because it is complete and optimal when consistent heuristics is provided. [2] In pathfinding is usually used euclidean distance.

2.3 AgentFly

AgentFly[11] is a multi-agent platform for simulation of air traffic that integrates features like collision avoidance, cooperative negotiation. It respects given aircraft's properties (BADA - Base of Aircraft Data[7]) and the planning uses maneuvers respective to current aircraft - e.g. minimum turn radius, maximum ascend and descend, speed, acceleration.

2.3.1 AgentFly path planner

There currently exists serial AA* (*Accelerated A**) path planner that uses adaptive step size depending on state's distance to the nearest obstacle [12]. There is a test for direct visibility to the goal performed for each state, allowing the search to quickly reach the goal in vast open spaces. This reduces number of visited states by the search and thus memory and time requirements. The heuristics is direct distance to the goal.

2.4 Parallelism

While A* is a serial (nonparallel) algorithm it cannot be used on multicore machines as is. The issue is that the search task is not possible to be fairly divided into parts for each thread to run autonomously because the state graph is not known before and is explored dynamically. Very often it is also infinite.

2.4.1 Overhead

Synchronization The *synchronization overhead* is the idle time wasted at synchronization points when some threads have to wait for others due to locking.

Search When expanding states in parallel, more states get expanded than would be required by serial search (explained in 4.4).

Communication Time spent by manipulation with states (e.g. forwarding to other thread) that wouldn't be done in serial search.

Time *Time overhead* is caused by expensive (or excess) I/O operations.

It is not uncommon that some parallel approach performs worse than serial due to these factors.

Chapter 3

State of the Art

3.1 PA*

The simplest parallel modification of A* is based on work stealing [6]. There is one CLOSED and one OPEN set. The basic A* algorithm is run n-times (A Centralized Parallel Search Strategy).

- This strategy has not much redundant search over A*.
- On the other side, the issue could be very frequent locking resulting in huge synchronization overhead and the approach could lead to serial-like performance.

3.2 HDA*

In HDA* (Hash Distributed A*) each thread owns a partition of the search space. The partitioning is done by a hash function k on the state [5].

Each thread **T** has its own CLOSED and OPEN set. The "A* inside **T**" then selects highest priority state for expansion from its local OPEN set. Expanded state **S** is then sent to other thread according to $k(\mathbf{S})$.

Each thread has a message queue, for receiving states from other threads, that is asynchronous and thus non-blocking.

Before **T** selects new state to expand it checks its message queue. Then for each received state **S'** it checks for duplicates in the local CLOSED set and accordingly inserts **S'** into OPEN set.

The most important is the choice of a hash function **k** so that all threads have approximately equal amount of work (states) assigned, and each thread has opportunity to reach the goal (e.g. no thread searches unpromising partition of the search space).

3.3 PBNF

Parallel Best-*N*Block-First unlike previous methods builds an abstract graph of nodes called *n*blocks that contain states from original domain [1]. If two states are successors in the

original domain than their *n*blocks are successors in the abstract graph too. Each *n*block has its own open and closed list (like in A*).

*N*blocks themselves are organised in a priority queue according to their best open node.

Thread acquires a free (not locked and any of its successors locked) *n*block with highest priority and locks it. Then it could expand its nodes and insert into appropriate *n*block.

No synchronization is needed except for acquiring an *n*block.

Chapter 4

Analysis

The path finding algorithm is to be used for planning trajectory of UAVs in A-globe [10] framework (AgentFly project). This requires the implementation to be programmed in Java. The search space is continuous. There exists a serial planner based on A*.

For parallelisation it is necessary to divide the search space so that every thread can explore its local scope. That can be done by a hash function that assigns a state to a thread according to its coordinates.

PBNF's main advantage is to reduce locking-related overhead. A graph of scopes is created according to their adjacency. While that is not trivial in continuous space, because the rules for adjacency cannot be done quickly (while in combinatorial problems this could be promising approach) and the resulting graph would be too fuzzy because some maneuvers can be extended to even nonadjacent (physically) scopes (e.g. long straight maneuver), PBNF is not considered in this work.

4.1 HDA*

HDA* was chosen because of its similarity to A* (serial BFS), and there is the only necessity to provide a hashing function.

The idea is that more threads can process more nodes in the same time that serial A* needs to process only one state. On the other hand, the speedup is reduced by search overhead (suboptimal nodes are processed) and communication overhead.

Each worker owns its defined portion of the search space, which is determined by the hash function. This allows checking for duplicates in the worker's closed set, because duplicate state always arrives to the same scope.

Generated states are sent as asynchronous messages to appropriate workers (determined by hash function), there is no need for synchronization. Each worker, before fetching the best state from open set, handles received states. It is checked for presence in closed set and *a*) inserts the received into local open set; *b*) or discards it, in case it has been already processed. Figure 4.1 shows the path of newly generated node (solid lines) to its destination worker's open set.

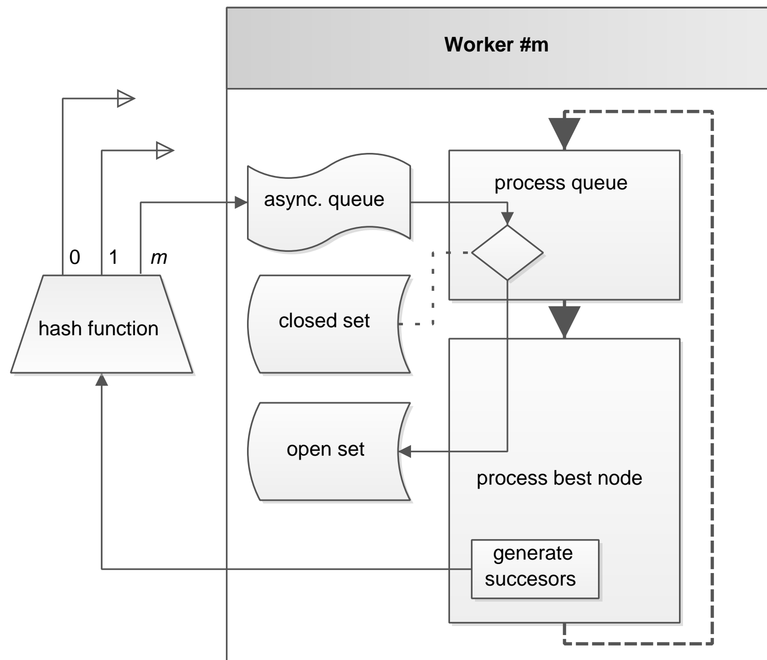


Figure 4.1: Manipulation with generated nodes in HDA*

4.2 PA*

There are possible slight modifications of location of structures that allow the search to proceed according to PA* approach:

- When the *closed set* is stored as a global structure (shared by all workers),
- and the *open set* is also shared structure. The generated states are inserted, without regard to the hashing, in the moment they are generated (this makes some synchronization overhead).

Furthermore, when only the *closed set* is shared, there is possible to distribute states from continuous space where two states' *equality* is implemented as *similarity*. This reduces some search overhead of HDA* in such domains.

The sharing is done by passing references to the global sets, that are given to the workers, instead of creating new sets. The operation of a single worker is similar to serial A* (Figure 4.2).

While there is no asynchronous transfer of the states, the open set is required to be thread-safe. This is done by synchronization, and that is the bottleneck of PA*, causing the search run serially in a very concurrent environment.

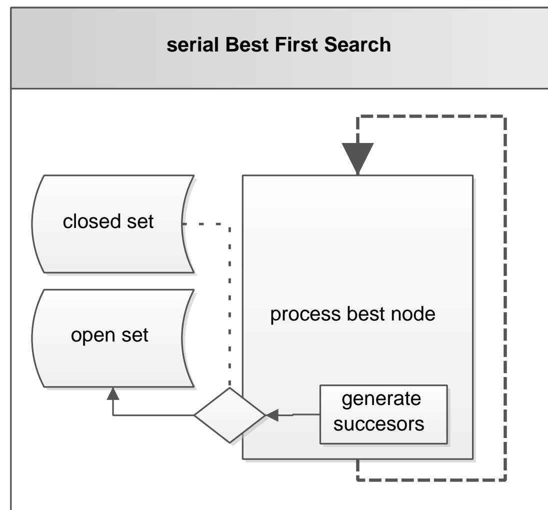


Figure 4.2: Manipulation with generated nodes in A*

4.3 Master-Worker

The search is provided by a master (as a Java method call). The master performs initialization (5.3.2.1) – main task is to spawn workers, each into separate thread. Then the master waits for the workers to end and collects the result.

When an exception occurs in a worker the master has to stop other workers and throw the exception into the main (calling) thread.

4.4 Optimality

While A* is guaranteed to find optimal solution when consistent (monotone) heuristics is provided [2], that doesn't account for any parallel approach.

The issues are

- When each thread searches defined portion of the search space, then it ignores other states/paths leading to its portion. That means it has to expect from other threads to provide better[1] instaces (duplicates) of any local state anytime.
- When all threads search the same (whole) search space, they necessarily expand suboptimal paths. That is because 1st thread takes the best state and meanwhile it processes it the nth thread takes n^{th}-best state. That breaks the basic A* idea and can generate some state earlier than it would be generated from some better predecessor.

That requires to compare new states against closed set (5.1.3.1) instead of just checking presence. That results in increased *time overhead* (2.4.1).

[1]having lower g – reached by a shorter path from the start

Also the **termination condition updates**: The search is finished when all threads' best states in open set are not better than the best goal found[2]. This result in *search overhead* (2.4.1).

4.5 Interface

Here it is described how I defined the minimal requirements for a programmer to implement if one wants to use BFS.

Following pseudocode describes A*'s operation.

```
1  function A*(start,goal)
2      closedset := empty set
3      openset := {start}
4
5      while openset is not empty
6          current := the node in openset having the lowest f value
7          if current = goal
8              return current
9
10         remove current from openset
11         add current to closedset
12         for each neighbor in neighbor_nodes(current)
13             if neighbor in closedset
14                 continue
15
16             if neighbor not in openset
17                 add neighbor to openset
18
19     return failure
```

Listing 4.1: A* pseudocode

There are highlighted fragments that are *inputs* for the A* search, some of them are functions.

It is clear that we need to provide

1. initial state (start)
2. comparator for states (finding *best node from openset*)
3. expansion function (*neighbor_nodes(current)*)
4. goal (*description* of desired state)
5. equality test for checking presence in sets

And HDA* adds up

6. hash function for distributing states

I tried to keep these requirements as general as possible. So there is no need to expose any *heuristics score* or *goal state instance*. Instead a way to compare two states and test for a goal, respectively, is required.

[2]For single-worker configuration this is equivalent to the standard A* termination condition, because A* selects always the best node for processing, so there are no better nodes implicitly.

Chapter 5

Implementation

The main part of this chapter describes a generic search approach that can be used for any state exploring task. The AgentFly planning is implemented afterwards on top of this core.

5.1 Best First Search

Serial BFS algorithm was described in 2.2 and 4.5. ***BFS.java*** is its abstract implementation. Its features are also used (inherited) in parallel classes.

The basic Java interface would look like

```
1 public abstract class BFS<T> implements Comparator<T>
2 {
3     final public T find(T start);
4     public abstract int compare(T state1, T state2);
5     protected abstract Iterable<T> expand(T state);
6     protected abstract boolean isGoal(T state);
7 }
```

The usage for a specific implementation *Impl* for states of type *StateType* is

```
1 class Impl extends BFS<StateType> {
2   ... // implementation of methods required
3 }
4 StateType start = ...;
5 StateType goal = new Impl().find(start);
```

It is obvious, that the

1. initial state is a parameter of `find()` call
2. comparator for states is the class *Impl* itself (**implements** Comparator<StateType>)
3. expansion function is part of *Impl*'s body
4. goal test is *Impl*'s predicate method (isGoal(StateType))
5. equality test is implemented as `equals(Object)` on the state.

The goal state (isGoal(s)==**true**) is returned from the find() call. When the search exhausts all reachable states and goal is not found, then **NULL** is returned.

All further described configurations and extensions are optional and are not required to be implemented for basic BFS (A).*

5.1.1 Detachment from the state's properties

Even in serial search it was necessary to calculate state's hashCode for use in closed set implemented as a HashMap for fast access. To make this independent on the state's type there can be found following code:

```
1  public abstract class BFS<T> implements Comparator<T>
2  {
3     ...
4    protected int hashCode(T state)
5      {
6         return state.hashCode();
7      }
8
9      protected boolean equals(T state1, Object state2)
10     {
11        return state1.equals(state2);
12     }
13     ...
14 }
```

These are proxy methods[1] to the original state.

It is useful when it is found out that equals should be different/complicated/cached etc to override them according to the implementation needs.

5.1.2 ClosedSet, OpenSet

These are interfaces based on specific needs of the search and can ba also custom-provided by the programmer (will be shown later).

```
1      public static interface BFS.ClosedSet<T>
2      {
3         public void add(T obj);
4         public T get(T obj);
5      }
6
7      public static interface BFS.OpenSet<T> extends BFS.ClosedSet<T>
8      {
9         public T peekTheBest();
10        public T pollTheBest();
11        public boolean isEmpty();
12     }
```

There exists basic implementation in the class that is suitable for most cases.

[1] Their use is in default BFS implementation enforced by using BFS<T>.StateHolder wrapper

The *ClosedSet* is backed by a *HashMap* (and respects proxy methods for equals() and hashCode(), see 5.1.1) to provide constant-time performance[2].

The *OpenSet* has a *HashMap* as in *ClosedSet* and a *PriorityQueue* to provide theBest() and isEmpty() methods. The peekTheBest() method has constant-time performance and $O(log(n))$ performance for modification methods[3].

Custom implementations can be provided by overriding following methods

```
1 protected ClosedSet<T> createClosedSet();
2 protected OpenSet<T> createOpenSet();
```

5.1.3 Configuration

The BFS class has a **protected** BFS.Configuration config property. The *Configuration* class has only property **boolean** checkClosedSetIfBetter.

Configuration properties are to be set in constructor:

```
1 class Impl extends BFS<StateType> {
2   public Impl()
3   {
4     this.config.someDirective = true/false;
5     // OR
6     this.config = SOME_CONFIGURATION_INSTANCE;
7   }
8 }
```

5.1.3.1 Checking closed set

When a new state s is generated it is checked whether closed set already contains such a state (that has equal hashCode(s) and equals(s,closedSet.get(s))).

(a) When the closed set **doesn't contain** s, s is inserted into open set for inspection.

(b) When the closed set **contains** state equal to s then one of the following actions is taken

  (b1) if checkClosedSetIfBetter = **false** *(default)* then s is discarded

  (b2) if checkClosedSetIfBetter = **true** then s is inserted into *open set* only if s is better than s' (where $s' \leftarrow closedSet.get(s)$), otherwise is discarded

This behaviour is required when

- the heuristics used for organising *open set* is not consistent
- or the expansion is done in parallel 4.4

otherwise the search wouldn't be optimal.

The reason for this is that a state **S** is guaranteed to be accessed by an optimal path on the graph only when the heuristics is consistent (Triangle inequality) [2]. Otherwise **S** could be accessed prematurely due to confusing (*defective*, non-consistent) heuristics.

[2] HashMap (Java Platform SE 6). <http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html>. [Online; accessed 9. 4. 2012]

[3] PriorityQueue (Java Platform SE 6). <http://docs.oracle.com/javase/6/docs/api/java/util/PriorityQueue.html>. [Online; accessed 9. 4. 2012]

5.1.4 Search interrupt - Timeout

Because the search could be *a)* infinite (e.g. the reachable search space is unbounded and the goal is unreachable) or *b)* time-critical (required to end within given time period), there is way to terminate the search:

```
public T find(T start, BFS.InterruptTest interrupt) throws
    InterruptedException
```

where the InterruptTest is an **interface**

```
public static interface InterruptTest
{
    public boolean doEnd();
}
```

When the doEnd() method returns *true* the BFS.find() immediately throws an InterruptedException.

For the most common case – time limiting, there is a BFS.InterruptTimeout class that will end the search after specified timeout in milliseconds elapses since the *InterruptTimeout* object creation.

```
try{
  long nTimeoutMillis = 1000; // 1000ms = 1s
  BFS.InterruptTest interrupt = new BFS.InterruptTimeout(nTimeoutMillis);
  T goal = BFSImpl.find(start, interrupt); // won't run longer than 1s
  ... // search ended properly
}catch(InterruptedException ex){
  ... // timeout expired
}
```

This approach allows to interrupt the search on the basis of any different signal, for example:

```
BFS.InterruptTest interrupt = new BFS.InterruptTest()
{
  public boolean doEnd()
  { // when plane crashes there is no need to plan path anymore
    return hasPlaneCrashed();
  }
}
```

5.1.5 Listeners

The BFS class can register listeners for one event – end of execution.

```
public void addExecutionFinishedListener(ActionListener listenter)
```

The listener's actionPerformed() is called when search exits. Either a goal is found or goal is not found (5.1) or the search is interrupted (5.1.4).

In this moment it is expected to do a cleanup of resources used for the purpose of the search.

It is recommended way to dynamically[4] register code that releases some expensive-to-create objects to their pool in the moment of their creation. E.g. huge array in open/closed set, database connections etc.

5.1.6 Statistics

The BFS class has several properties begining *log...* for statistics of the search. They can be accessed by getters.

logOpenProcessed
: count of states processed in main A* loop (tested for goal, expanded, ...)

logGenerated
: count of all states returned by expand()

logGeneratedOpenReplaces
: nodes that have been generated, were already contained the openset, and were re-inserted

logGeneratedBetterAlreadyInOpen
: nodes that have been generated, were already contained the openset, and were thrown away

logGeneratedCloseReplaces
: nodes that have been generated, were found in closed (but with worse score), and were re-insterted into open (5.1.3.1 - when configured not to check closed set this will be 0)

logGeneratedAlreadyInClose
: generated nodes found in close that were thrown away for this fact

logRuntimeMillis
: wall time in milliseconds (available after execution finishes)

5.2 AgentFly Path Planner

The serial planner's (2.3.1) code was separated into methods to fit the interface already described in this chapter; the open and closed sets were reimplemented and some other minor modifications were done to allow concurrency. This allows to run the same code in parallel (PBFS, described further).

5.3 Parallel Implementation

For parallelization I chose approach described in [5]. The main idea is to provide an asynchronous queue for each thread that is used to deliver states expanded by other threads.

Destination thread for a state is to be determined by a hash function $k(\mathbf{S})$.

Each thread's loop is extended:

[4] considered it is not known in advance whether some object will be created or instances are created dynamically

1. While state queue of this thread is not empty: for each state check whether its duplicate is in closed set[5] and appropriately insert into local open set.

2. Process best state from open set ... (as in BFS 4.5). New states $\mathbf{S}_1..\mathbf{S}_n$ are asynchronously sent to their threads' queues according to $k(\mathbf{S}_i)$.

5.3.1 Implementation with regard to BFS

I required the extension from BFS to PBFS (parallel best first search) to be implemented in Java as effortlessly as possible. Therefore PBFS extends BFS.

The only method that must be implemented extra is `clone()`, the reason is described later in 5.3.2.2.

PBFS also introduces constructor with one integer parameter - number of threads to run. When created using nonparametric constructor then `Runtime.getRuntime().availableProcessors()` is used.

5.3.2 Master - Worker lifecycle

Further will be used terms

Master that is basically the object visible from the calling scope and it's `find()` provides the result. It is instance of PBFS.

Worker is of type PBFSWorker and is spawned by master. Each worker is run in separate thread and passes goal states to the master (`PBFS.foundGoal()`). Worker implements `Runnable` interface.

5.3.2.1 Initialization

On the `find()` call on the master following actions are taken (in the master's thread):

1. For each $i = 1..n$ an asynchronous queue (`ConcurrentLinkedQueue<T>)waitingForOpen[i]` and (`PBFSWorker<T>)workers[i]` are created.[6]
2. For each $i = 1..n$ is a new thread created. Worker `workers[i]` is run in this thread.
3. Wait until all workers initialize (generally create open and closed set).
4. Give a start signal for workers to enter search loop.
5. Wait for the search to end.
6. Process exceptions (optional).
7. Return solution.

In Java it is:

[5] Described in 5.1.3.1, explained in 4.4
[6] n - number of threads

```
1 public T find(T start, final BFS.InterruptTest interrupt) throws
    InterruptedException
2 {
3   ...
4   currentBest = null;
5   for (int i = 0; i < numThreads; i++)
6   {
7    waitingForOpen[i] = new ConcurrentLinkedQueue<T>();
8    PBFSWorker<T> worker = new PBFSWorker<T>(this, start, interrupt, i);
9    workers[i] = worker;
10  }
11  Thread[] threads = new Thread[numThreads];
12  for (int i = 0; i < numThreads; i++)
13  {
14   threads[i] = new Thread(workers[i], this.getClass().getSimpleName() + " 
       worker #" + i);
15   threads[i].start();
16  }
17  initSignal.await();
18  startSignal.countDown();
19  doneSignal.await();
20  if (m_throwable != null)
21  {
22   if (m_throwable instanceof InterruptedException)
23      throw (InterruptedException) m_throwable;
24   else
25      throw new RuntimeException(m_throwable);
26  }
27  return currentBest;
28  ...
29 }
```

Listing 5.1: PBFS.find()

5.3.2.2 Master cloning

Worker has the A* loop inherited from BFS and most of other methods (especially for manipulation with states) are proxies to the master or to *master's clone.*

```
1 public PBFSWorker(PBFS<T> master, T start, BFS.InterruptTest interrupt, int
    index)
2   {
3      super();
4      this.master = master;
5      this.implementationClone = master.clone();
6      this.start = start;
7      this.interrupt = interrupt;
8      this.index = index;
9      this.config = this.master.config;
10  }
```

Listing 5.2: PBFSWorker's constructor

Methods expand(T), isGoal(T), compare(T,T), hashCode(T), and equals(T,Object) are called on the implementationClone.

Reason for this is that when those method would be called on one instance of the master, from any worker at any time, they would need to be thread safe[7]. With the cloning it is necessary just to create clone of required parts and the methods' implementation remains single thread.

In case the master is not needed to be cloned (is thread-safe), write

```
protected PBFS<T> clone()
{
  return this;
}
```

Then implementationClone==master for each worker.

5.3.3 Configuration

In PBFS is configuration of type PBFS.Configuration **extends** BFS.Configuration, having following properties.

5.3.3.1 Sharing closed set

Depending on **shareClosedSet** value. When

false ***(default)***
: closed set is created for each worker.

true
: all workers share the same closed set. Can be useful when a custom implementation is provided such that doesn't use the same hashCode() (and equals(), consequently)[8] as is used for states distribution.

In both cases result of createClosedSet() call is used (5.1.2). Yet in the latter case, it is required to return instance of PBFS.ConcurrentClosedSet<T>.

5.3.3.2 Sharing open set

Depending on **shareOpenSet** value. When

false ***(default)***
: open set is created for each worker.

true
: all workers share the same open set. When they take the best state from it, it may be in fact n^{th}-best, because another worker might have taken those better ones already. Similar implementation as for closed set (see above) is required.

[7] Or **synchronized** but that would cause the workers calling them to execute serially. On the other hand, when a proper clone is provided, then the methods can be **synchronized**. Although that is not necessary, because they would be called from single thread only.

[8] e.g. when states are considered *equal* when similar – so the already closed state would be in different worker's scope

5.3.3.3 Sequential distribution

Depending on **allocateWorkSequentially** value. When

false ***(default)***
: a hash function *hashCode* is used on a state s to be assigned to worker ($hashCode(s) \bmod n$).

true
: the state is according to the order i in which it was generated sent to worker ($i \bmod n$).

5.3.4 ClosedSet, OpenSet

As mentioned in 5.3.3.1, PBFS.ConcurrentClosedSet<T> may be required (and PBFS.ConcurrentOpenSet<T>). These interfaces doesn't introduce any new functionality from BFS.ClosedSet<T> (its ancestor), it is just to make the implementing programmer aware of the concurrent nature.

Drawback: The set's type is checked in the runtime depending on current configuration. When it is not appropriate, a UnsupportedOperationException is thrown.

The default PBFS's implementation uses internally Java's ConcurrentHashMap for the closed set. This supports some level of concurrency.

For open sets PBFS provides static method synchronizedOpenSet() to wrap single-thread implementation.

```
1 protected OpenSet<T> createOpenSet()
2 { // returns PBFS.ConcurrentOpenSet<T>
3   return PBFS.synchronizedOpenSet(new SomeSingleThreadOpenSetImpl());
4 }
```

5.3.5 Synchronization

To satisfy the terminating condition in parallel environment (4.4) it is necessary for the workers to wait for each other sometimes to detect whether to continue search. The naive approach is to process one state (in each thread) and then come together and see, what happened.

That unfortunately leads to enormous overhead - the workers would wait most of the time for the slowest one (who got some state difficult to expand, or who's thread is not running).

My implementation of a worker does work while it has work to do. That means while there are states in my open set that are better than current (globally) best goal (or if the open set is not empty in case, there has not been found any goal yet).

When there is no work at this moment that doesn't mean to finish, because there shall be some states, from other workers, for me to be generated. My thread yields (to enable other threads run), and makes a quick test whether other workers work. That is done by calling master.isContinueSearch().

```
1 if allFinished
2   return false;
3 else if workingWorkersCount > 0 // workers that isWorking==true
4   return true;
5 else if statesInMessageQueuesCount > 0
6   return true;
7 else
8   return false;
```

Listing 5.3: master.isContinueSearch() in pseudocode

When **true** is returned, the worker starts over, and either does this active waiting loop again, or has some states to process.

When **false** is returned, there is chance that the (whole) search has no other candidates for a goal, and master.isEveryWorkerFinished() is called.

```
1 public boolean isEveryWorkerFinished()
2 {
3     if (allFinished)
4        return true;
5     allFinishedLock.lock();
6     try
7     {
8      // wait until all threads meet in this method
9        while (!allFinished && allFinishedLock.getQueueLength() != numThreads -
             1)
10       {
11           if (isContinueSearch(-1))
12           { // polling whether is some work already
13              return false;
14           }
15           Thread.yield();
16       }
17       // endwhile => all threads met here
18       allFinished = true;
19       return true;
20    } finally
21    {
22       allFinishedLock.unlock();
23    }
24 }
```

Listing 5.4: master.isEveryWorkerFinished()

This ensures that the search terminates properly only when all workers agree that there is no other work (meaning no other promising states).

This is shown in figure 5.1.

The reason for these two tests is that the final one is quite expensive - it uses locking. While the first one is asynchronous (therefore quick) and, in case the search is not over, sufficient. It just is not trustworthy when it says “do not continue search”.

There exists a allFinished flag allows to terminate the search immediately when set to **true**, its purpose is described later.

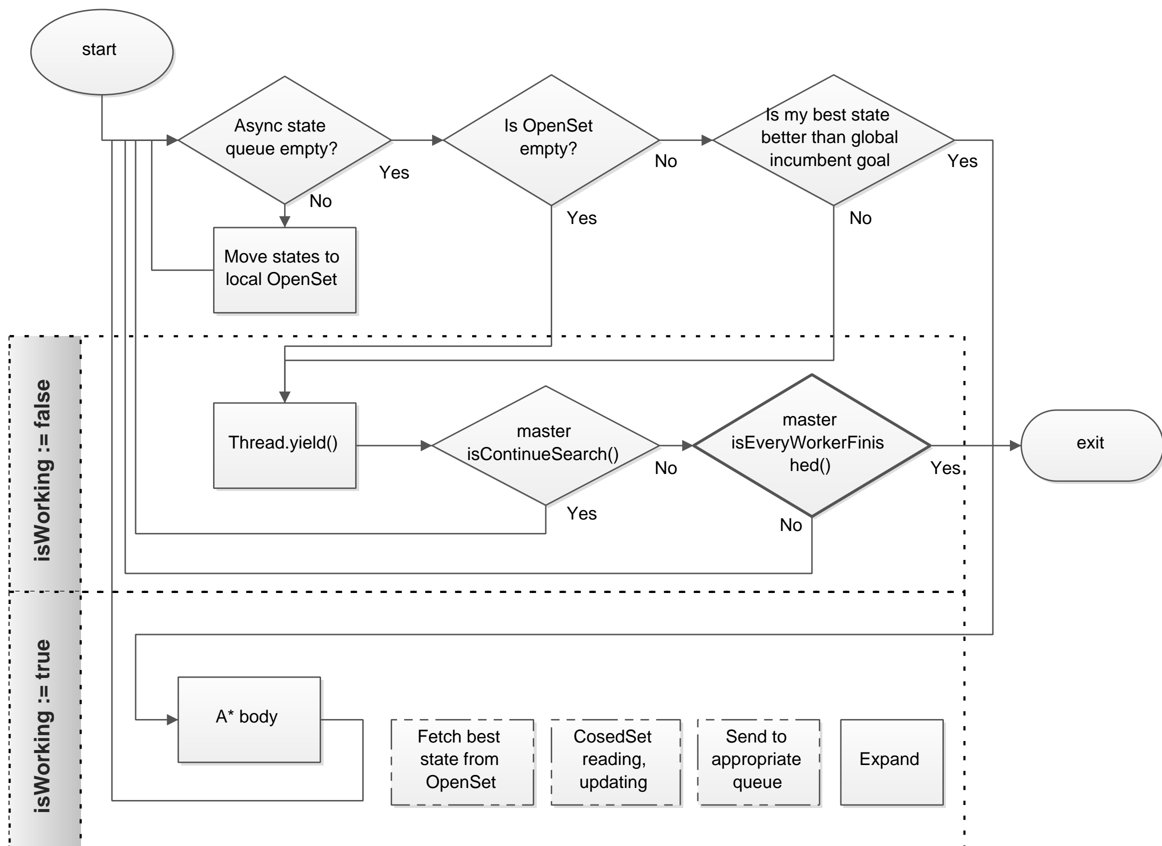


Figure 5.1: Operation of a worker

5.3.6 Exception handling

When a exception in a worker is thrown the master is notified.

- allFinished is set to **true**, causing all other workers to end immediately,
- it is labeled by it's thread name (containing worker number and implementing class),
- when all workers terminate it is thrown from the PBFS.find() call – as an InterruptedException in case the interrupt test was positive, or wrapped as a RuntimeException otherwise.

This makes the program to end up with exception with verbose stack trace

```
1 java.lang.RuntimeException: java.lang.Exception: Exception in thread "BFSTest worker #1"
2     at PBFS.find(PBFS.java:121)
3     at BFS.find(BFS.java:31)
4     at BFSTest.main(BFSTest.java:34)
5 Caused by: java.lang.Exception: Exception in thread "BFSTest worker #1"
6     at PBFS.workerException(PBFS.java:186)
```

```
7     at PBFSWorker.run(PBFSWorker.java:160)
8     at java.lang.Thread.run(Thread.java:662)
9 Caused by: java.lang.ArithmeticException: / by zero
10    at BFSTest.fib(BFSTest.java:96)
```

Listing 5.5: Example stack trace

5.4 Configuration Drilldown

Depending on configurations flags (5.1.3 and 5.3.3) the search can operate in several modes.

5.4.1 HDA*

checkClosedSetIfBetter	true
shareClosedSet	false
shareOpenSet	false
allocateWorkSequentially	false

Each worker gets its own closed and open set. Distribution of the states is done by hash function k and states are delivered to their destination worker $k(\mathbf{S})$ via non-blocking queue.

This requires to provide hash function k such, that it distributes states fairly and all workers have approximately same number of promising states.

5.4.2 HDA* with shared closed set

checkClosedSetIfBetter	true
shareClosedSet	true
shareOpenSet	false
allocateWorkSequentially	false

Closed set is created globally for all workers. The processed states are added concurrently and checking is also concurrent (5.3.4).

This has an advantage when ClosedSet organisation does not comply the k (states distribution) and states for a worker to process are not necessarily stored in the worker's closed set.

5.4.2.1 Sequential distribution

checkClosedSetIfBetter	true
shareClosedSet	true
shareOpenSet	false
allocateWorkSequentially	true

Sharing of the closed set relieves the necessity to have fixed scopes that ensure that when a state is generated twice it goes to the same worker.

This approach does not require the programmer to provide fair hash function k, because it doesn't influence the work distribution. Each worker gets assigned same amount of states to process.

5.4.3 PA*

checkClosedSetIfBetter	true
shareClosedSet	true
shareOpenSet	true

Only one open and closed set is maintained, like in serial A*. The difference is that many workers use them in parallel.

That causes *synchronization overhead* (2.4.1) especially for the reason of modification of (sorted) open set.

On the other hand, this has lower *search overhead* than previous configurations, because there are always taken the n-th **globally** best nodes to expand, while in HDA* there are taken the **locally** best nodes, that can be much worse.

5.4.3.1 Suboptimal PA*

checkClosedSetIfBetter	false
shareClosedSet	true
shareOpenSet	true

As described in 4.4, the parallel implementation requires to compare states in the the closed set to remain optimal, this accounts also for PA*. But unlike HDA*, PA* doesn't deviate much from the serial-A*'s order of processed states.

In most cases, PA* is able to find optimal path without comparing states to closed set and especially in continuous search space, there is always some tolerance because the search space is relaxed into discrete approximation.

This can run a bit faster than PA*, and may be suitable for some problems.

Chapter 6

Experimental Results

The planning was implemented and tested for AgentFly path planner and Sliding Tile Puzzle domain. The algorithm was run in Java 6. The Java Virtual Machine (JVM) was provided 40GB of memory and run on 2x Intel Xeon E5620 (2.40GHz) in Windows 7 64bit. Each problem was solved by serial A*, PA* and HDA* in up to 16 threads. Every combination was invoked 30 times to provide statistically significant results.

To allow approximately the same conditions for each measurement, each configuration was run before the actual measurement took place. This should have warmed-up the JVM. Specifically to load all classes required, to perform additional just-in-time compilation, inlining code, and adaptive optimization. Before each measured planning System.gc() was called, making the JVM to reclaim unused objects from the memory.

6.1 AgentFly Path Planner

Two scenarios were used for the benchmark. The first was Planner setup test (no.4, from predefined configurations, figure 6.1), that contains two obstacles and thus doesn't take advantage of the heuristics, especially in the beginning where f leads to explore the dead end. The second scenario is planning in 3D mountainous terrain that contains three independent segments, each is planned separately (figure 6.2). As a hash function was used state's mainHashCode, that reflects it's position in 3D space.

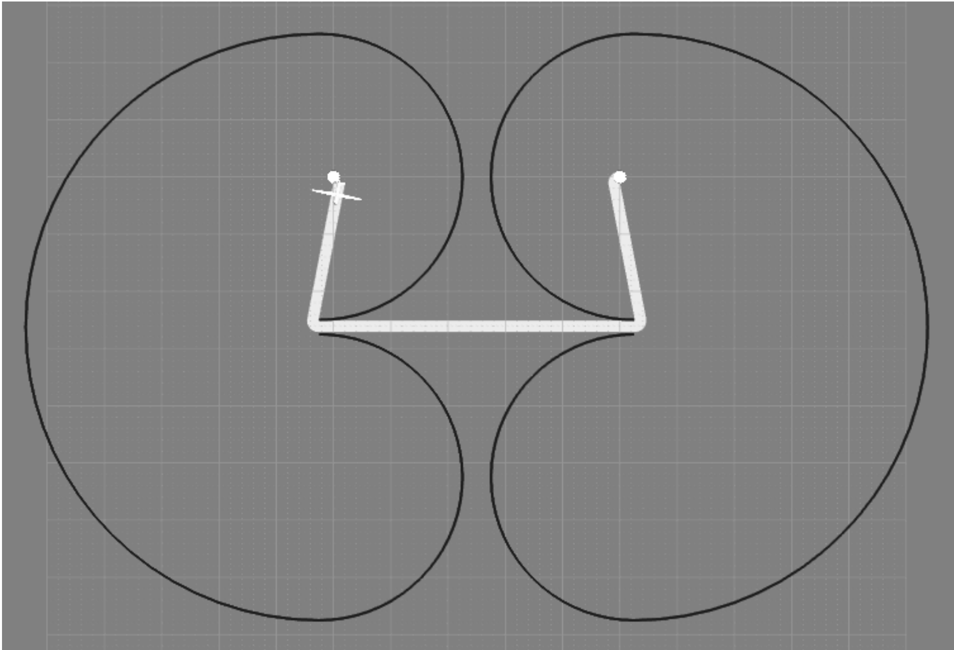


Figure 6.1: Planner setup test 4

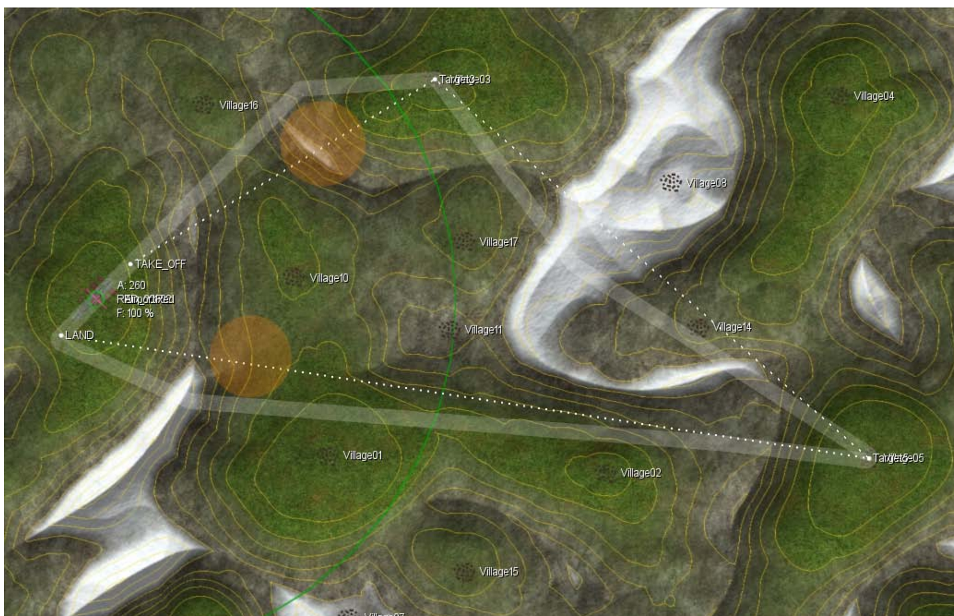


Figure 6.2: Mountain

6.2 Sliding Tile Puzzle

The parallel search was also implemented to solve 15-Puzzle game with intent to compare the nature of these two domains on the empirical evaluation results. The problem can be viewed as a planning of consequent moves that lead to final board configuration.

The board is represented as flat integer array with values 1–15 describing tile position, and zero stands for the gap. Each expansion represents one move of a tile onto the place of the gap, making g increase by one. The heuristics is sum of a manhattan distances for a tile to its desired place for each tile. A Java's built-in hash for integer array is used for hashing.

For testing a problem was chosen that requires about 250 thousand states to be visited by serial A* (figure 6.3).

1	2	6	4
5	3	7	14
	9	15	8
11	13	12	10

Figure 6.3: The 15-Puzzle task

6.3 Search Time and Volume

For each task was measured wall time[1] depending on thread count (Figure 6.4, left side). There were also collected data showing search overhead (the right side of figure 6.4, specifically number of processed nodes.

6.4 Speedup Over Serial A*

Figure 6.5 shows the speedup of the overall search (left) and amount of nodes processed in a second over serial A* (right).

[1] measuring starts when find() is called and ends when master returns solution

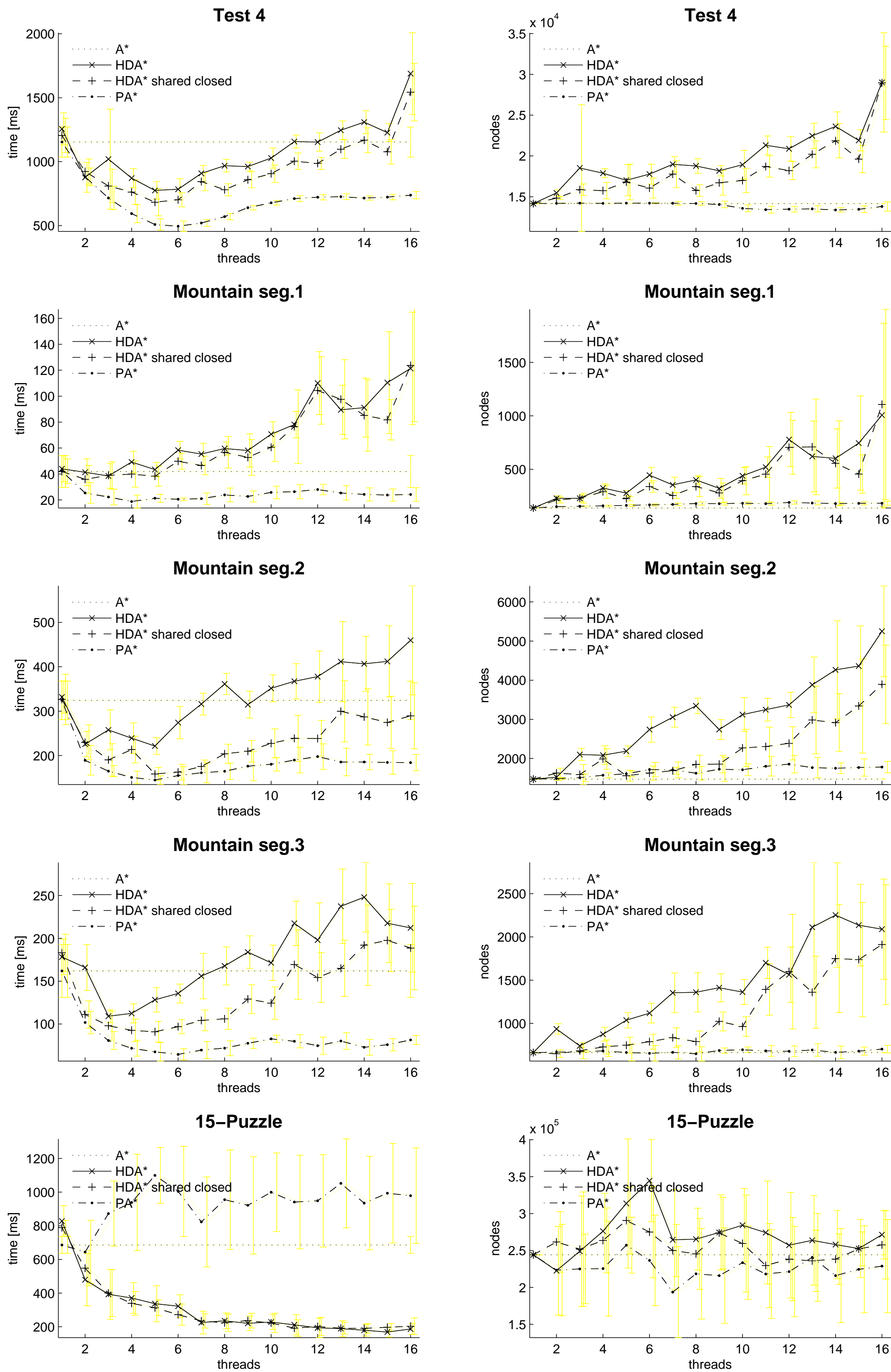


Figure 6.4: Results

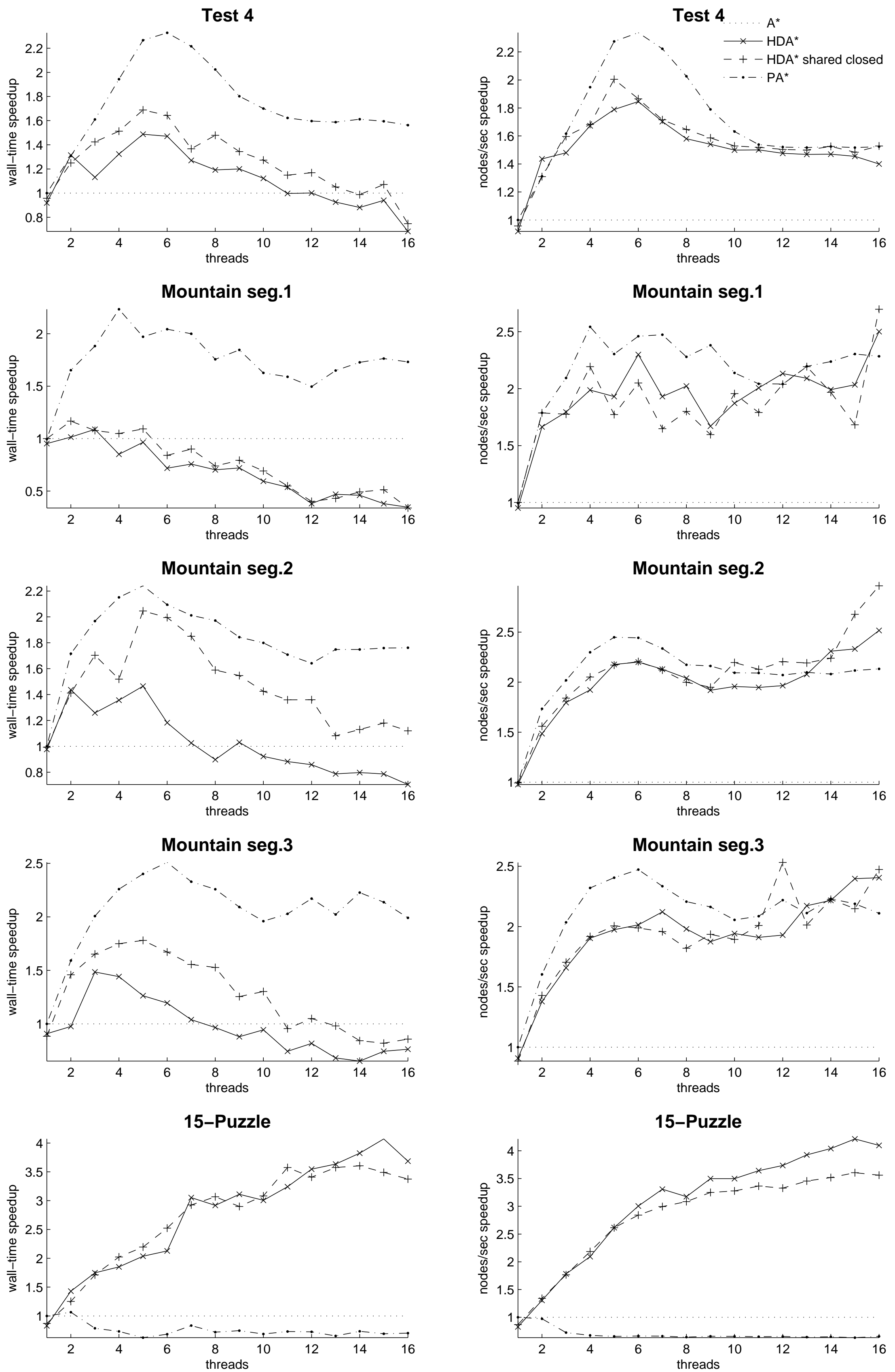


Figure 6.5: Speedup over serial A*

Chapter 7

Discussion

The results have shown that for AgentFly path planning the parallel extension generally performs worse than for 15-Puzzle task, yet still can perform better than serial A* when suitable configuration is used. From the graphs (figure 6.5) it can be seen, that the speedup grows mostly in the beginning (few threads running). Later, when more threads are added, the growth declines.

7.1 Observed Issues

7.1.1 Increase of search overhead

In figure 6.4 it can be seen, that for the AgentFly path planning tasks, the number of processed nodes is increasing with threads, unlike for the 15-Puzzle – there the number of processed nodes stays in the bounds of serial A* approach. It was expected behaviour, because HDA* and PA* takes advantage of speculative expansion of suboptimal (or locally optimal) nodes. In the path planning, the heuristics guides the search to the goal after much less nodes expansions than in the 15-Puzzle, where the search has to visit many indistinguishably promising states. This has high impact on the final speedup. When the search overhead outnumbers parallel speedup of nodes processing, the search runs slower than serial A*.

7.1.2 Hashing in continuous space

Continuous space is usually handled as a discrete space where *near* states are considered as equal. This could lead to a situation, when such *equality* isn't transitional, and therefore states couldn't be always hashed to preserve equivalence classes in each scope. This allows duplicities in disjoint scopes.

Such duplicates can be eliminated by sharing closed set; that also allows concurrency (thus doesn't introduce significant synchronization overhead).

The search overhead is visible in figures 6.4 where there are always more nodes processed in HDA* over *HDA* with shared closed set* for the path planning tasks.

7.1.3 Increase of communication overhead

Considering following situation: Node s_1 is current globally best, and leads to the goal optimally. It's (s_1's) successor will therefore be some node s_2, that is also on the final path. Figures show A*[1] (figure 7.1) and HDA* (7.2) progress.

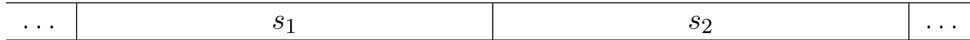


Figure 7.1: A* timeline

When the A* algorithm is finished processing state s_1, it immediately chooses s_2 for further processing (in the same thread).

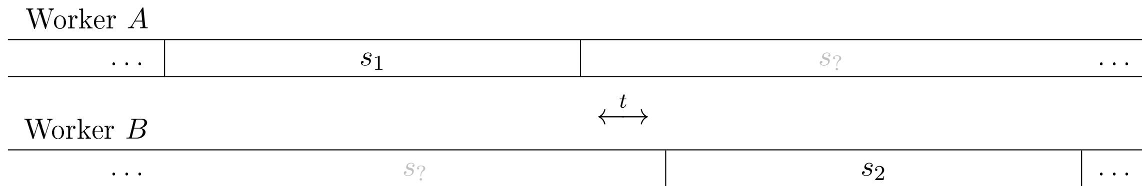


Figure 7.2: HDA* timeline

Whereas in HDA*, worker A (who is processing s_1's scope) sends s_2 to worker B. When B finishes some state he is currently processing, he chooses s_2. That delay (marked t in the figure) could last up to the time required for processing one state, but could be also insignificant (approximately zero). In average case, it would lead up to 50 % slowdown of the search.

This delay is an issue only for nodes that are on the path to the goal, that are to be processed just after they have been generated, and thus their delayed processing makes the search slow down. It's impact depends on how much is the search straightforward.

In fact, the delay only approaches 50 % as more threads (workers) are running. Because there is n^{-1} chance[2], that state s_1 belongs to the same worker's scope as s_2 does, and there would be no delay. More accurate formula for the worst case is $50 \cdot (1 - n^{-1})\%$ slowdown.

7.1.4 Increase of synchronization overhead

Figures 6.5 show that, for AgentFly tasks, the search slows down to a certain level when many threads are running. It is caused by few synchronized parts of code for expanding a state. These synchronized fragments make the search, in very concurrent environment, run serially. While there is some search overhead due to suboptimal nodes expansion, the search can be even slower than serial A*.

[1] applies to serial A* and PA*, that has some overhead (due to more advanced data structures), but the main principle is the same

[2] expecting that fair (balanced) hash function is used; n is number of workers

7.2 AgentFly Path Planner

Results show that PA* is always the fastest for AgentFly planning. The observed search overhead is *none*, PA* visits as many states as A* does. For lower number of threads (up to 6) there is increase in performance.

HDA* is exploring the same number of states per second as the PA*, but the search overhead increases accordingly and in more threads it is beaten by A*. HDA* with shared closed set is faster, because it doesn't suffer from duplicates in disjoint scopes (7.1.2). This reduces the search overhead.

The unnecessary search overhead results from the nature of path planning. The heuristics often works very well and the search doesn't retract much to suboptimal states and most of the time are processed nodes that have just been generated. The heuristics (and the other tricks, including adaptive step size; 2.3.1) leads to the goal after very few nodes processed. That makes no significant advantage of speculative expansion in workers' local scopes.

The PA*'s bottleneck should be the synchronization while accessing the open set, but in this case, it takes advantage of relatively time-consuming state expansion (checking for collisions, calculating angles etc.) that is much more expensive than synchronization on shared open set; that holds mainly for lower number of concurrent threads.

It can be seen (figure 6.5) that for larger number of threads (six) the planner is not processing more nodes at a time. This is due to overhead caused by synchronization, in some internal methods, related to zone checking in the AgentFly planner.

7.3 15-Puzzle

HDA* shows speedup up to 4 times faster than serial A* in 16 threads (figure 6.5). There isn't any difference in sharing closed set, because hashing totally reflects equality test. The speedup is direcly proportional to the amount of nodes processed per second.

On the other hand, PA* goes 10 % slower than serial A*, independently on number of the threads. The reason is the synchronization overhead when manipulation open set, combined with huge number of visited states ($2.5 \cdot 10^5$, meaning state expansion is relatively cheap compared to synchronization on open set).

Chapter 8

Conclusion

Parallel search algorithms for planning were implemented in this work. The goal was to achieve speedup in planning tasks on multicore computers. The basic concept is to process more nodes than does serial search in the same time, allowing the search to finish faster. To distribute work among threads HDA* or, alternatively, PA* approach is used.

The implementation was done in Java™ programing language as a generic search framework. Communication among threads in the HDA* is done using asynchronous message queues, thus without synchronization overhead. The PA* is based on work-stealing and requires synchronization on key structures. The search performance was tested on AgentFly path planning, and additionally 15-Puzzle game.

Empirical results have been documented in several benchmarks, where the parallelization proved its capability to accelerate states processing. The tasks were run in up to 16 threads. There have, however, been observed several issues affecting overall speedup. Those depend on specifics of search domains and the implementation.

The AgentFly path planning was accelerated only slightly more than 2 times over serial A*. This was caused by synchronization in some methods for checking correctness of a path. The PA* was preferable to HDA*, because HDA* introduces search overhead, that wasn't counterbalanced due to the synchronization.

For the 15-Puzzle task there was significant speedup (up to 4 times) and the figures hint it would still increase with more threads. This is possible due to the implementation, that was non-blocking, and huge number of states was required to be processed in this task.

8.1 Future Work

For the path planner, there should be proposed expansion method that doesn't require synchronization. This will enable to scale well even for many threads.

Another speedup can be achieved by bidirectional search – to run another search from the end to the start, expecting to meet the forward one [8].

Bibliography

[1] E. Burns, S. Lemons, R. Zhou, and W. Ruml. Best-first heuristic search for multi-core machines. In *Proceedings of the 21st international joint conference on Artifical intelligence*, pages 449–455, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[2] 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.

[3] HashMap (Java Platform SE 6). <`http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html`>. [Online; accessed 9. 4. 2012].

[4] PriorityQueue (Java Platform SE 6). <`http://docs.oracle.com/javase/6/docs/api/java/util/PriorityQueue.html`>. [Online; accessed 9. 4. 2012].

[5] A. Kishimoto, A. Fukunaga, and A. Botea. Scalable, parallel best-first search for optimal sequential planning. In *Proceedings of the International Conference on Automated Scheduling and Planning ICAPS-09*, pages 201–208, Thessaloniki, Greece, 2009.

[6] Vipin Kumar, K. Ramesh, and V. Nageshwara Rao. Parallel best-first search of state-space graphs: A summary of results. In *in Proc. 10th Nat. Conf. AI, AAAI*, pages 122–127. Press, 1988.

[7] A Nuic. User manual for the base of aircraft data (bada)-revision 3.1. *a a*, 2(3):106, 2010.

[8] I. Pohl. Bi-directional search. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, pages 127–140. Elsevier, New York, 1971.

[9] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, December 2002.

[10] David Šišlák, Martin Rehák, and Michal Pěchouček. A-globe: Multi-agent platform with advanced simulation and visualization support. In Andrzej Skowron, Rakesh Agrawal, Michael Luck, Takahira Yamaguchi, Pierre Morizet-Mahoudeaux, Jiming Liu, and Ning Zhong, editors, *Web Intelligence*, pages 805–806. IEEE Computer Society, 2005.

[11] David Šišlák, Přemysl Volf, Štěpán Kopřiva, and Michal Pěchouček. Agentfly: Nas-wide simulation framework integrating algorithms for automated collision avoidance.

In *Integrated Communications, Navigation and Surveilance Conference (ICNS)*, pages F7–1 – F7–11. IEEE, May 2011.

[12] David Šišlák, Přemysl Volf, and Michal Pěchouček. Flight trajectory path planning. In Neil Yorke-Smith Luis Castillo, Gabriella Cortellessa, editor, *Proceedings of ICAPS 2009 Scheduling and Planning Applications woRKshop (SPARK)*, pages 76–83, Greece, September 2009.

Appendix A

CD Contents

```
text
     Contains this document

src/pbfs
     PBFS classes described in the Implementation chapter

src/atc/planner
     Classes for AgentFly parallel planner implementation
     that comply with up-to-date[1] AgentFly project

src/puzzle
     15-Puzzle PBFS implementation and simple testing task launcher
```

[1] May 15, 2012