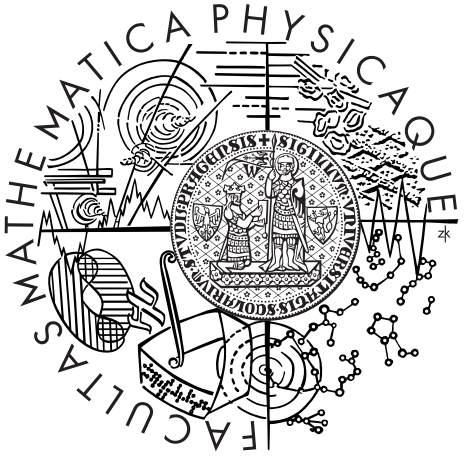
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Martin Hanes

Petrobras Planning Domain: PDDL Modeling and Solving

Department of Theoretical Computer Science
and Mathematical Logic

Supervisor of the bachelor thesis: Mgr. Daniel Toropila
Study programme: Computer Science
Specialization: Programming

Prague 2012

I would like to dedicate this thesis to the memory of my grandfather, Ján Adam, the most determined, passionate and loving person in my life.

I would like to acknowledge the assitance of Mgr. Daniel Toropila, who has been very generous in his support of my academic pursuits and has given me many helpful ideas, feedback and advice. I would like to thank my family and my friends for their support, tolerance and empathy.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Plánovací doména Petrobras: Modelování PDDL a řešení

Autor: Martin Hanes

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Daniel Toropila

Abstrakt: Práce se zabývá doménou Petrobras definovanou pro soutěž ICKEPS 2012. Popisuje příklad těžkého problému, který stojí na hranici mezi rozvrhováním a plánováním. Popisuje state-of-art techniky moderní umělé inteligence, které byly vybrány pro tento účel. Dále práce provází čtenáře procesem modelování v doméně Petrobras v jazyce PDDL a vysvětluje výsledky získané existujícími nástroji používanými v rámci komunity zabývající se plánováním. V závěru experimentálně srovnává výsledky zvoleného přístupu a ostatních přístupů používaných v soutěži ICKEPS 2012, které poukazují na použitelnost existujících sekvenčních plánovacích systémů pouze na malé instance problémů z řešené domény.

Klíčová slova: klasické plánování, modelování domény, PDDL, logistická doména

Title: Petrobras Planning Domain: PDDL Modeling and Solving

Author: Martin Hanes

Department: Katedra teoretické informatiky a matematické logiky

Supervisor: Mgr. Daniel Toropila

Abstract: The thesis explores Petrobras domain from the challenge track of the ICKEPS 2012 competition, which is an example of a difficult problem standing on the borderline between planning and scheduling. It describes state-of-art techniques of the modern artificial intelligence that were chosen for this purpose. It guides the reader in the process of modeling the Petrobras domain in PDDL language and explains the results provided by the existing tools used within planning community. In the end it compares the results of the chosen approach to the other approaches used in ICKEPS 2012 competition, showing that the existing sequential planning systems are usable only for solving smaller problem instances of this domain.

Keywords: classical planning, domain modeling, PDDL, logistics domain

Contents

1. Introduction

Sections in Introduction chapter briefly introduces Petrobras problem, automated planning and scheduling and how they are related. This chapter gives a reader a quick overview of the bachelor thesis.

1.1 Insight into Petrobras problem

Petrobras is a semi-public Brazilian energy and petroleum company, one of the five largest in the world and the largest in Latin America due to 2011 revenues [sta09]. It proposed the *Petrobras problem* (*PP*) for the ICKEPS 2012, International Competition on Knowledge Engineering for Planning and Scheduling[1] as planning and scheduling problem. It can be described as the need to provide transportation of goods and tools from two ports on the land to platforms in the ocean located in strips. These strips are divided into two parts called *Rio de Janeiro* and *Santos*. There are six platforms in Rio de Janeiro strip and four in Santos. Transport is done by vessels which start and finish at one of the *waiting areas* in the ocean. The problem to be solved is to process a list of requests of goods delivery and assign vessels to load cargo items from port(s) and deliver them to platform(s) in the ocean. Schedule, the result, should be optimized in several ways: vessels should carry as much cargo items as possible, use as little fuel as possible and of course makespan[2] should be as short as possible. Also the *docking cost* and number of vessels should be minimized.

1.2 What is automated planning and scheduling?

Planning is an abstract, explicit deliberation process that chooses and organizes actions by anticipating their expected outcomes. This deliberation aims at achieving as best as possible some prestated objectives. Automated planning is an area of Artificial Intelligence that studies this deliberation process computationally[NGT04]. It takes a description of the initial state of the world, a description of the desired goals and a description of a set of possible actions. Then it attempts to find a plan that is guaranteed to generate a set of actions execution of which leads from any of the initial states to one of the goal states.

Planning and scheduling (P&S) are closely related problems. In a simple decomposition scheme, planning appears to be an upstream problem that needs to be solved before scheduling. Planning focuses on the causal reasoning to find the set of actions needed to achieve the goal, while scheduling concentrates on time and resource allocation for this set of actions.

Scheduling addresses the problem of how to perform a given set of actions using a limited number of resources in a limited amount of time. A resource is an entity that one needs to borrow or to consume (e.g., a tool, a machine, or energy)

[1] Official webpage: http://icaps12.icaps-conference.org/

[2] In manufacturing, the time difference between the start and finish of a sequence of jobs or tasks.

in order to perform an action. An action may have the choice between several alternate resources, and a resource may be shared between several actions. The main decision variables for scheduling a given action are which resources should be allocated to the action and when to perform it[NGT04].

There are many approaches on planning, we can choose between domain dependent (*ad-hoc*) planning and *domain independent* planning, between *online* and *offline* planning, *classical* and *temporal* planning, whether the *search* algorithm is exploring *state-space* or *plan-space* and so on. Every approach is useful for particular problem and it is important to choose one of them carefully when starting to solve planning problem.

It is natural to divide problems into categories and adapt specific representations and algorithms for each one. What they all have in common, is that we always need to *search* the space that is exponential and so the time complexity of the algorithms used is harder than NP-complete.

1.3 Planning and scheduling and Petrobras problem

The task here is to carry out a research in the P&S area in order to find out, which tools can be used to solve PP, which techniques we should use and what will the representation look like. We need to keep in mind the objectives - what we need to optimize.

The problem is well defined and easy to imagine. Motivation here is not only to participate in the ICKEPS competition, but also the fact that it is a real problem and any thoughts and ideas about solving it are appreciated.

2. First thoughts and overview

Chapter reviews steps followed in solving PP and organization of the thesis.

2.1 Steps followed to solve Petrobras problem

Solving temporal planning problems optimally is difficult. Moreover, there are not too many temporal planning systems available for the use. However, for the cases when we are interested mostly in optimizing the used resources, such as consumed fuel, one of the possible approaches is to try solving the corresponding sequential planning problem (with numerical fluents) where the action durations are ignored, and where the amount of the used fuel is to be minimized. Not only we are then to solve a simpler problem, but there is also a plethora of successful sequential planning systems that we could employ for this purpose. Once we then have a solution for the simplified problem ready – a sequential plan – we can reincorporate back the action durations, concurrent resource constraints and causal dependencies between the actions, and based on that information we can schedule the actions by assigning their start and finish timestamps, obtaining thus the required valid temporal plan.

The first step thus was to obtain a valid sequential plan. Second chapter describes how a model of the sequential PP was composed and how sample problems in *PDDL language* were generated.

The second step was to find out what the constraints and dependencies between actions are, so we are able to turn a sequential plan into parallel one. Third chapter explains the theory of this process and clarifies the algorithm used.

The last steps were to compute costs of parallel plans, generate a series of problems and process the results, and finally, if possible, to compare the results with other solving systems.

2.2 Overview of chapters in this work

The first information about PP and automated planing and scheduling can be found in the **introduction** of this thesis.

Chapter two (this one) briefly summarizes how the solving was done and why this method was chosen. It also gives an overview of chapters of this work and its organization.

Chapter three is about modeling a problem and modeling language PDDL.

3. Modeling a domain

This chapter describes modeling in automated planning and scheduling world, PDDL, the most used modeling language, and shows how Petrobras domain was modeled.

3.1 Why do we need a common modeling language?

One of approaches to the P&S is to create a domain-independent model for planners and schedulers. Using the same modeling language we are able to solve any kind of problem as long as we are able to create a model, which represents that problem. The problem (its model) is encoded in modeling language and processed as input by solver. Solvers, on the other hand, can be benchmarked. Their performance can be compared by running the same set of problems and by comparing their results.

3.2 What is PDDL?

PDDL, *Planning Domain Definition Language*, inspired by STRIPS and ADL was first developed in 1998 by Drew McDermott and his colleagues. It was an attempt to standardize planning languages in order to make IPC[1] 1998 possible[Fou12c].

Since 1998, PDDL evolved from PDDL 1.2 to PDDL 3.1 mainly because of the IPC competition needs[Hel11]. Every version enriches the previous one with new features. With a few minor exceptions, the old PDDL version is subset of the new one, so a planner developed for some version of PDDL as input is also compatible with older versions. There is no explicit declaration of PDDL version in source code PDDL file, but it contains *requirements* clause, where all of the requirements used in domain are listed. For example *strips*, *typing* or *fluents*. The PDDL version can also be defined as the smallest superset which defines all of the listed requirements.

The model of a problem in PDDL consists of a *domain definition* and a *problem definition* and is typically separated and saved as two files.

3.3 Modeling Petrobras

Obviously, the first decision we need to make is to choose what PDDL version we are going to use. As was said in the previous text, the PDDL version is defined by a set of requirements (features we use for modeling) listed in domain pddl file. So we need to determine which requirements are necessary and which are just helpful and we are able to substitute them or simply leave them out.

From my experience, a planner checks a list of requirements in a PDDL domain file and if it finds a requirement that it does not support, it exits with error. On

[1] International Planning Competition, web: http://ipc.icaps-conference.org/

the other hand, if PDDL domain file is defined without *:fluents* requirement but it uses fluents in *:functions* or *:actions*, or it is defined without *:typing* requirement but it uses types, a planner usually parses whole input file without error and even without warning.

Quite essential requirement, from my point of view, is *:typing*, which is a tree-structured system of object types, similar to the classic object inheritance in C-based languages[2]. It limits predicate's atoms to certain types. For example we are able to define types:

Listing 3.1: Example of types definition

```
(:types
    location vessel - object
    platform port waiting-area - location
)
```

and then use single-typed predicate:

Listing 3.2: Example of predicate with typing

```
(:predicates
    ;; vessel is at location (platform, port or waiting area)
    (at ?vessel - vessel ?where - location)
)
```

instead of triple:

Listing 3.3: Example of predicates without typing

```
(:predicates
    ;; ?obj1 is at ?obj2
    (at ?obj1 ?obj2)

    ;; ?obj is a vessel
    is_vessel(?obj)

    ;; ?obj is location
    is_location(?location)
)
```

PP is defined as optimization problem so our attempt is to find the best possible plan, according to some metrics. In other words, we are interested in how "good" the solution is, not only if "any" solution exists. The costs of actions are not uniform, so we need to use PDDL with *:action-costs* requirement to be able to set different costs to different actions. Of course, we need to find a planner which supports it.

[2]C, C++, C#, Java...

The last requirement is *:numeric-fluents*. First versions of PDDL supported only binary state variables. With numeric fluents we are able to use numerical (real-valued) state variables. In PDDL 3.1 *:object-fluents* were introduced[Has11], so state variables can be mapped to a finite domain. The *:fluents* requirement stands for *:numeric-fluents* and *:object-fluents* (NOTE: some of the planners are able to accept *:fluents* requirement but they reject *:numeric-fluents* despite the *:numeric-fluents* is subset of *:fluents*).

3.3.1 Domain definition

This section describes how the domain part of PDDL was analyzed and modeled.

Typing

We used object typing as follows:

Listing 3.4: Typing used

```
(:types
    location vessel cargo - object
    logistics_location waiting-area - location
    platform port - logistics_location
)
```

It is clear that every object is a descendant of the object type. *Vessel* and *cargo* types are types of vessel and cargo objects. The *location* type is common type for all the platforms, ports and waiting-areas, and represents every place a vessel can navigate to. We need to make a difference between locations where cargo items can be loaded and unloaded (platforms and ports) and where they cannot (waiting areas). Therefore, the *logistics_location* is parent type for *platform* and *port* types.

Predicates

All predicates - used by a planner as boolean state variables - are listed in the *:predicates* section of domain in PDDL:

Listing 3.5: Predicates used

```
(:predicates
    ;; location is waiting area
    (is-waiting-area ?loc - location)

    ;; vessel is at location (platform, port, waiting area)
    (at ?vessel - vessel ?where - location)

    ;; vessel is docked in port or platform
    (is-docked ?v - vessel ?where - location)

    ;; platform is able to refuel a vessel
    (platform-can-refuel ?platform - platform)
```

```
13
14      ;; cargo is at location
15      (cargo-at ?c - cargo ?where)
16
17      ;; helping predicates
18      ;; vessel was once docked at this location (navigate action
            negates this predicate)
19      (vessel-once-docked-at-location ?v - vessel ?l - location)
20
21      ;; vessel was once refueled at location (undock action
            negates this predicate)
22      (vessel-once-refueled-at-location ?v - vessel ?l -
            logistics_location)
23  )
```

(is-waiting-area ?location) A location is a waiting area (WA). It is used in *navigate-empty-vessel* action, to differentiate between WA and two other location types. If a vessel travels to WA, it needs enough fuel to get from that place to the nearest refuel location.

(at ?vessel ?location) A vessel is physically in a location. It is used as precondition for navigate actions and a dock action.

(is-docked ?vessel ?location) A vessel is docked at a platform or a port. Although it seems to be useful to use *logistics_location* type instead of *location*, the second parameter cannot be of the *logistics_location* type. That is because this predicate is used in navigate action - a vessel can *not* be navigated when it is docked. Parameter *?from* of the navigate action is, of course, of the type *location* (vessel can be navigated from any location, not only from *logistics_location*).

(platform-can-refuel ?platform) In Petrobras domain specification[Igr12], it is mentioned that both ports and *some* of the platforms can refuel a vessel. This predicate informs which platforms are able to do refueling.

(cargo-at ?cargo ?object) Intuitively, this predicate is used when cargo is located at some *object*. The *object* type is common for both *location* and *vessel* types.

(vessel-once-docked-at-location ?vessel ?location) The predicate indicates that a vessel was docked once at a dock or a platform. It ensures that a planner does not dock a vessel more than once at the same location if it does not navigate between the docking actions. In other words, a vessel should be navigated to another location in order to be docked again. It seems to be intuitive, that the second variable of this predicate could (and should) be of *logistics_location* type, which is more strict than *location* type, because we cannot dock at waiting areas. However, it is necessary that the second variable of this predicate is of *location* type, because it is one of the *:effects* of the navigate actions.

(vessel-once-refueled-at-location ?vessel ?logistics_location) Similarly, this predicate ensures that a vessel is refueled only once during docking.

Functions

Functions in PDDL map predicates to an object or number (fluent). They are used as (world) state variables by a planner.

Listing 3.6: Functions used

```
(:functions
    ;; the sum of fuel used by all the ships - to be minimized
    (total-fuel) - number

    ;; the free space in a vessel
    (vessel-free-capacity ?vessel - vessel) -  number

    ;; how many vessels can be docked in location
    (free-docks ?loc - logistics_location)  - number

    ;; max free capacity of a vessel
    (max-vessel-free-capacity) - number

    ;; current tank of a vessel
    (fuel-level ?vessel - vessel) - number

    ;; max fuel level (after refueling) independent on vessel
    (max-fuel-level) - number

    ;; the tank consumption - depending on the state of vessel (
       empty or not)
    (navigation-cost-empty ?from ?to - location) - number
    (navigation-cost-nonempty ?from ?to - location) - number
    (navigation-cost-empty-to-nearest-refuel-loc ?waiting-area -
       location) - number

    ;; the distance between from and to locations
    (distance ?from ?to - location) - number

    ;; the weight of a cargo
    (cargo-weight ?c - cargo) - number
)
```

(total-fuel), (fuel-level ?vessel), (max-fuel-level) functions are used to model total amount of fuel used by all the vessels, current fuel level of a vessel and the fuel capacity of a vessel.

(vessel-free-capacity ?vessel) and (max-vessel-free-capacity) indicates how many tons of cargo are currently loaded onto a vessel and the maximum sum of weights of cargo items that can be loaded onto a single vessel at the time.

(free-docks ?logistics_location) is a number of currently available docking places in a location.

(navigation-cost-empty ?from ?to - location) is number of liters of fuel used to navigate with an empty vessel between locations.

(navigation-cost-nonempty ?from ?to - location) is the same as previous, with a nonempty vessel.

(navigation-cost-empty-to-nearest-refuel-loc ?location) is number of liters needed to travel from a waiting area to the nearest location, which is capable of refueling.

(cargo-weight ?cargo) defines the weight of a cargo item in tons.

(distance ?from ?to - location) is never used in a problem definition file. It is, however, used in post-processing to determine the distance between locations.

Functions:

- *(max-fuel-level)*
- *(navigation-cost-empty ?from ?to)*
- *(navigation-cost-nonempty ?from ?to)*
- *(navigation-cost-empty-to-nearest-refuel-loc ?waiting_area)*
- *(cargo-weight ?cargo)*
- *(distance ?from ?to)*

are read-only. They are defined in advance in a PDDL problem file and never changed by any action. Therefore, we can think of them as constants defining the problem case.

On the other hand, *(total-fuel)*, *(fuel-level ?vessel)* and *(vessel-free-capacity ?vessel)* are defined in PDDL problem file and their values are changing during the planning process.

- *(total-fuel)* is increased by every navigate action
- *(fuel-level ?vessel)* is decreased by every *navigate* action and increased by the *refuel* action
- *(vessel-free-capacity ?vessel)* is decreasing by a *load* action and increased by an *unload* action

Operators

Listing 3.7: Actions used

```
;;navigate vessel if it's empty
(:action navigate-empty-vessel
    :parameters (?vessel - vessel ?from ?to - location)
    ;;<listing omitted>
)

;;navigate vessel if it's NOT empty
(:action navigate-nonempty-vessel
    :parameters (?vessel - vessel ?from ?to - location)
    ;;<listing omitted>
)

;;vessel loads cargo at location
(:action load-cargo
    :parameters (?vessel - vessel ?c - cargo ?loc -
       logistics_location)
    ;;<listing omitted>
)

;;vessel unloads cargo at location
(:action unload-cargo
    :parameters (?vessel - vessel ?c - cargo ?loc -
       logistics_location)
    ;;<listing omitted>
)

;; vessel refuels at refueling platform
(:action refuel-vessel-platform
    :parameters (?vessel - vessel ?p - platform)
    ;;<listing omitted>
)

;; vessel refuels at port
(:action refuel-vessel-port
    :parameters (?vessel - vessel ?p - port)
    ;;<listing omitted>
)

;; docks vessel in port or platform
(:action dock-vessel
    :parameters (?vessel - vessel ?where - logistics_location)
    ;;<listing omitted>
)

;; undocks vessel from port or platform
(:action undock-vessel
    :parameters (?vessel - vessel ?where - logistics_location)
    ;;<listing omitted>
)
```

navigate-empty-vessel and navigate-nonempty-vessel actions navigate a vessel between locations. As a side-effect, they set *(vessel-once-docked-at-location ?vessel ?location)* predicate to false, so a vessel can be docked in the next location. We use two types of navigate actions to differentiate between navigating empty vessel and non-empty vessel. The metrics description in Petrobras domain specification[Igr12] says that the fuel consumption and the duration of a navigate action is influenced by weight of cargo carried by a vessel and that we use only two states of a vessel - empty and non-empty. In other words, there is no difference between a vessel carrying only one ton of cargo and fully-loaded vessel.

refuel-vessel-platform and refuel-vessel-port actions refuel a vessel. Changing a *(vessel-once-refueled-at-location ?vessel ?location)* predicate, it ensures that vessel is refueled at most once during a single docking.

load-cargo and unload-cargo actions manipulate with cargo items to and from a vessel. They increase and decrease a corresponding *(vessel-free-capacity ?vessel)* function.

dock-vessel and undock-vessel actions use a *(free-docks ?logistics_location)* function to ensure the constrained number of vessels in a location. It is also guaranteed that vessel is docked/undocked at most once before navigating to another location. As a side-effect, the *undock-vessel* action sets a *(vessel-once-refueled-at-location ?vessel ?where)* predicate to false.

The whole domain definition PDDL file is listed in Appendix B.

3.3.2 Problem definition

In this step we model a problem using object types, predicates and functions defined in the domain definition PDDL file. We define actual objects, an initial world state and a goal. It is natural that no actions are defined or redefined here.

Objects

Listing 3.8: Example of the objects declaration

```
(:objects
    P1 P2 - port
    F1 F2 F3 F4 F5 F6 G1 G2 G3 G4 - platform
    A1 A2 - waiting-area
    V1 V2 V3 - vessel
    C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 C11 C12 - cargo
)
```

Ports, platforms and waiting-areas part is basically the same for all the problem instances. Petrobras domain specification[Igr12] defines two waiting areas (A1, A2), two ports (P1, P2), six platforms in Rio de Janeiro strip (F1 - F6), and four platforms in Santos strip (G1 - G4) and a distance between every pair

of locations. As we can see from the listing above, there is an object defined for every vessel and for every cargo item.

Initial world state

The *:init* section describes an initial world state (WS). Basically, it could be defined as a set of predicates and functions (functions in a PDDL context, see section 3.3.1 that are valid at a very first WS.

The following example shows how the first part of *:init* section may look like:

Listing 3.9: Example of the first part of a *:init* section

```
1  (= (total-fuel) 0)
2  (= (max-fuel-level) 600)
3  (= (max-vessel-free-capacity) 100)
4
5  (= (free-docks F1) 1)
6  (= (free-docks F2) 1)
7  (= (free-docks F3) 1)
8  (= (free-docks F4) 1)
9  (platform-can-refuel F5)
10 (= (free-docks F5) 1)
11 (= (free-docks F6) 1)
12 (= (free-docks G1) 1)
13 (= (free-docks G2) 1)
14 (platform-can-refuel G3)
15 (= (free-docks G3) 1)
16 (= (free-docks G4) 1)
17 (= (free-docks P1) 2)
18 (= (free-docks P2) 2)
19
20 (is-waiting-area A1)
21 (is-waiting-area A2)
```

As we can see, the first line declares that the sum of used fuel is 0 in the initial WS. The second line sets the maximum fuel level of all the vessels. We have already mentioned that this function acts as a read-only variable and is used in refueling actions. Likewise, the *(max-vessel-free-capacity)* function is read-only and defines maximum weight of cargo that a vessel is able to carry. Lines 5 to 21 are standard functions and predicates, valid and unchanged for all problems in Petrobras domain. The *(free-docks ?logistics_location)* function is initially set to 1 for all the platforms and to 2 for all the ports. All waiting areas are listed in *(is-waiting-area ?location)* predicate.

In the initial WS, the only difference between a pair of vessels can be which waiting-area they anchor in. However, we need to enumerate all of the 3 properties of a vessel: where it is anchored, what the fuel level is and its loading capacity. Listing below shows an example of initial state definition for 3 vessels.

Listing 3.10: Example of a *:init* section for vessels

```
(at V1 A1)
(= (fuel-level V1) 600)
(= (vessel-free-capacity V1) 100)

(at V2 A2)
(= (fuel-level V2) 600)
(= (vessel-free-capacity V2) 100)

(at V3 A1)
(= (fuel-level V3) 600)
(= (vessel-free-capacity V3) 100)
```

Next, every cargo item is fully defined by two properties: its location and its weight. The following listing shows an example of initial state definition for 3 cargo items:

Listing 3.11: Example of a *:init* section for cargo items

```
(cargo-at C1 P2)
(= (cargo-weight C1) 25)

(cargo-at C2 P2)
(= (cargo-weight C2) 11)

(cargo-at C3 P1)
(= (cargo-weight C3) 6)
```

In the last part of the *:init* section we need to define distances and fuel costs for every pair of locations in the problem instance.

First, we define navigation costs to the nearest refuel station (location) for every waiting area:

Listing 3.12: *:init* section - navigation cost to the nearest refuel location

```
(= (navigation-cost-empty-to-nearest-refuel-loc A1) 24)
(= (navigation-cost-empty-to-nearest-refuel-loc A2) 20)
```

At last, we define *navigation-cost-empty*, *navigation-cost-nonempty* and *distance* functions for every pair of locations in the problem. The purpose of these functions can be found in section 3.3.1. Although the paths are symmetric, we need to define explicitly a distance and fuel consumption for both directions. So, for example, to define a path between ports P1 and P2 with a distance of 200 km, with a fuel cost 40 liters if it is empty and 67 liters if it is not, we write:

Listing 3.13: *:init* section - the distance and fuel consumption example

```
(= (navigation-cost-empty P1 P2) 40)
(= (navigation-cost-nonempty P1 P2) 67)

(= (navigation-cost-empty P2 P1) 40)
(= (navigation-cost-nonempty P2 P1) 67)

(= (distance P1 P2) 200)
(= (distance P2 P1) 200)
```

Goal

The *:goal* section defines a set of predicates, that are true for a goal world state. We achieve a goal state by executing actions found by a planner. In other words, if a plan (a schedule) is valid, we get from initial WS to one of the goal states by executing actions in their defined order. In PDDL, the goal WS is defined as a list of predicates and functions that are valid in the goal WS.
In Petrobras domain, the goal predicates are very simple: the finish location of every cargo item and the finish location of every vessel. Cargo location is defined by *(cargo-at ?cargo)* predicate. It is not strictly specified in which of the two waiting-areas a vessel should finish. So it is enough to satisfy at least one of the predicates (at ?vessel A1), (at ?vessel A2) assuming that objects A1 and A2 represent two different waiting-areas. Disjunction is achieved by *OR* operator. The following listing shows an example of a *:goal* section with 3 cargo items.

Listing 3.14: Example of a *:goal* section

```
(:goal
    (and
        (cargo-at C1 F4)
        (cargo-at C2 F4)
        (cargo-at C3 G4)

        (or
            (at V1 A1)
            (at V1 A2)
        )
        (or
            (at V2 A1)
            (at V2 A2)
        )
        (or
            (at V3 A1)
            (at V3 A2)
        )
    )
)
```

By default, a sequential planner optimizes a plan (a schedule) to contain as few actions as possible. On the other hand, a temporal planner usually optimizes

a plan to be as short as possible and uses durations of actions as a metric. Fortunately, it is possible to explicitly declare which function (PDDL function acts as a state variable) should be used as a metric during a planning process. Unfortunately, SGPlan is able to optimize only one function, so we chose the most important one and declared *(total-fuel)* as a metric for Petrobras domain.

Listing 3.15: *:metric* section

```
1 (:metric
2     minimize (total-fuel)
3 )
```

The *total-fuel* cost was described as a primary metric and it was used as the only metric in the *easy scenario* of the Petrobras problem. After the experiments are finished, we will find out if *total-fuel* metric correlates with other metrics and therefore it is enough to optimize only the fuel cost.

3.3.3 Experiences during modeling process

Modeling of Petrobras domain and problem instances was tested and few approaches were tried. From the beginning, SGPlan[HW06] was used as a primary planner. Intuitively, I have tried to trim the search space by limiting object types for actions as much as possible. It is important to say that the attempts to improve the computing time and costs by modifying PDDL files were made in time when I was having fully-functional system including modeling, scheduling and costs.

The first step was to create **nice problem** instance. It contains 3 vessels, each with 600 liters of fuel and 18 cargo items. I have chosen a name "nice" because it contains 18 cargo items which is quite a lot and also because SGPlan returned plan for the first version of the PDDL domain file. When SGPlan was run on that *first* version, it took 28 seconds to compute a sequential plan. The *nice problem* will be referred later in the thesis. The next step was to edit the first version of the PDDL domain file and create more versions and see how behavior of SGPlan changes and also to compare the results. The costs were saved for future comparison.

First, I have limited ?to, the last parameter of *(navigate-nonempty-vessel ?vessel ?from ?to)* action, to use only objects of *logistics_location* type instead of more general *location* type. The *location* type includes *waiting-area* type. Naturally, we do not need to navigate nonempty vessel to *waiting area*. Surprisingly, no plan was found using this modification so the more general *location* type is used. Moreover, if *(not (is-waiting-area ?to))* predicate was added to the preconditions list of the action, plan was not found too.

In order to ensure that a vessel refuels only once between *dock* and *undock* actions, we use *(vessel-once-refueled-at-location ?v - vessel ?l - logistics_location)* predicate. Also, we use *(vessel-once-docked-at-location ?v - vessel ?l - location)* to ensure that a vessel docks only once when anchored at a location.

The shortest distance between every two locations in Petrobras is the direct route. So to ensure that a vessel uses always the shortest path between two destinations, a new predicate *(vessel-once-navigated ?v - vessel)* was added, similar to the other two, described above. It was surprising, that SGPlan was unable to find a valid plan with this domain modification.
Also, modifying actions: *load-cargo*, *unload-cargo*, both *refuel* actions and *undock* action has redundant preconditions - *at* and *is-docked*. Obviously, *is-docked* implies *at* - a vessel is at location if it is docked there. However, the research and experiments have shown that if we remove *at* predicate from preconditions of these actions, SGPlan fails in finding a plan as a difficulty of a problem instance is increasing.

3.3.4 Problem generator in Java

In contrast to the domain PDDL file, which was designed and written once in a text editor, we need to generate a problem PDDL file based on the given parameters such as cargo list definition, number of vessels (and their starting location), vessel capacity and vessel fuel capacity. Moreover, we need to generate the cargo list and vessels list randomly for testing and benchmark purposes. Therefore, the random generating must be deterministic - by increasing the number of randomly generated cargo items by one the only change in a list of cargo items is *adding* one new cargo item to the previous list. In other words, the new list must *not* be randomly generated - only a single new item should be randomly generated and added to the previous list. This is achieved by simply adding a *seed* to the Java Random generator object[3]. Another advantage of this approach is that we need not to save generated PDDL files if we want to run a planner more than once on the same set of problems.

The generator is written in Java as a command-line program. It is wrapped-up using a simple shell script named *generate_pddl*.

Its parameters are (in this order):

- number of cargo items
- number of vessels
- max fuel capacity in liters
- the seed

Parameters are optional and the default values are 10 10 600 1.

[3] java.util.Random

4. Making parallel schedule from sequential plan

4.1 Overview

If a sequential planner (SGPlan) succeeds in finding a plan for given domain and problem PDDL files, it saves an output text file with a valid sequential plan. One example of such file is listed below:

Listing 4.1: SGPlan result example

```
1  ; Time 0.03
2  ; ParsingTime 0.02
3  ; NrActions 14
4  ; MakeSpan
5  ; MetricValue 277.000
6  ; PlanningTechnique Modified-FF(enforced hill-climbing search) as
       the subplanner
7
8  0.001: (NAVIGATE-EMPTY-VESSEL V3 A2 P2) [1]
9  1.002: (DOCK-VESSEL V3 P2) [1]
10 2.003: (LOAD-CARGO V3 C2 P2) [1]
11 3.004: (LOAD-CARGO V3 C1 P2) [1]
12 4.005: (UNDOCK-VESSEL V3 P2) [1]
13 5.006: (NAVIGATE-NONEMPTY-VESSEL V3 P2 F1) [1]
14 6.007: (DOCK-VESSEL V3 F1) [1]
15 7.008: (UNLOAD-CARGO V3 C2 F1) [1]
16 8.009: (UNDOCK-VESSEL V3 F1) [1]
17 9.010: (NAVIGATE-NONEMPTY-VESSEL V3 F1 F3) [1]
18 10.011: (DOCK-VESSEL V3 F3) [1]
19 11.012: (UNLOAD-CARGO V3 C1 F3) [1]
20 12.013: (UNDOCK-VESSEL V3 F3) [1]
21 13.014: (NAVIGATE-EMPTY-VESSEL V3 F3 A2) [1]
```

In order to create a valid parallel plan from this output we need to perform the following steps:

- parse the SGPlan output (result) file and create objects which represent actions
- parse the problem PDDL file to acquire details about the actions of the plan (e.g. the weight of cargo items cannot be obtained from SGPlan result file)
- set a duration to every action, based on its parameters
- generate a valid parallel schedule based on constraints and properties of the Petrobras domain

The command-line program called *ScheduleGenerator* was developed for this purpose. Again, Java was used, as it is one of the most used object-oriented multiplatform language. The representation of actions and objects in *ScheduleGenerator* is very similar to the *ProblemGenerator* - the program developed for

generating Petrobras problem PDDL files. Parsing input files for *ScheduleGenerator* is quite straight-forward. Also, computing a duration of every action based on its parameters is not very difficult. Petrobras domain specification[Igr12] clearly specifies how parameters of action influence its duration. The most difficult step is the last one. To find out what conditions need to be met for a pair of actions to be able to be executed parallel is not a very easy task.
The procedure of making a valid parallel plan, while achieving as short makespan as possible is discussed in the next section.

4.2 Theory

We started with a sequential plan - a list of actions in a given order. If the actions are executed, we get from the initial state to a the goal state. In the initial state, all of the cargo items are in their *starting* location and they are in their *finish* location in the goal state. These cargo items are transported by vessels, which are able to refuel during loading and unloading in some of the platforms and in all ports.

Delay what you cannot do right now

It is natural to think of a plan as a set of subplans, each subplan for one vessel. For example, if a plan uses two vessels - say V1 and V2, let us take every action linked to V1 and call this sequence Plan1 and every action linked to V2 and call this sequence Plan2. Then, we could start executing these subplans simultaneously, so the makespan is equal to the biggest makespan of all the subplans. However, if we look closer to this approach, we find out that this is not always possible, because preconditions of an action do not need to be met necessarily. For example, two or more vessels cannot be docked in a platform at the same time - we need to *delay* the docking action of one of the vessels to make a valid schedule. Another example is a scenario in which a vessel V1 transports an item from P1 to P2 and a vessel V2 transports the same item from P2 to its final location P3. If V2 is in P2 sooner than V1, the cargo item is not there yet. The action needs to be delayed. In conclusion, when generating a schedule, we need to check for every action, if its preconditions are met to execute it and if they are not, delay it.

Preserve the original order of actions in a subplan

The next question is: Can a vessel perform the actions in *any* order, if their preconditions are met? Of course, it cannot. Think of a refueling action: it has its purpose that a vessel refuels in specific moment, *before* or *after* some action(s). If we schedule a refuel action *after* some long journey instead of *before*, a vessel may not be able to get there. On the other hand, if we schedule it too soon, a vessel may not be able to make a whole journey again. Therefore, a sequence of actions in each subplan for each vessel need to be preserved.

Mutual exclusivity of a pair of actions

Generally, two actions are mutex[1], if their effects are inconsistent or if effects of one action is inconsistent with preconditions of another. Formally, action a and b are *not* mutex if:

$$\text{eff(a)}^+ \cap (\text{eff(b)}^- \cup \text{prec(b)}^-) = \emptyset \tag{4.1}$$

$$\text{eff(b)}^+ \cap (\text{eff(a)}^- \cup \text{prec(a)}^-) = \emptyset \tag{4.2}$$

$$\text{eff(a)}^- \cap \text{prec(b)}^+ = \emptyset \tag{4.3}$$

$$\text{eff(b)}^- \cap \text{prec(a)}^+ = \emptyset \tag{4.4}$$

In previous equations, *prec* and *eff* stand for preconditions and effects, + and - for positive and negative. The definition of preconditions and effects (and their negative forms) can be found in Automated planning and scheduling[NGT04].

Listed constraints seem to be strong enough to tell if a pair of actions can or cannot be parallel. However, we need to bear in mind constraints that arise from the domain itself. Previous constraints are only *necessary conditions* for a pair of actions to be parallel. For example, according to previous constraints, a pair of actions (LOAD V1 C1 P1) ≡ (loading cargo item C1 in port P1 on a vessel V1) and (UNLOAD V1 C2 P1) ≡ (unloading cargo item C2 in port P1 from a vessel V1) *could* be parallel. But from a Petrobras problem description, we know that a vessel cannot perform loading and unloading operations at the same time.
So we need to add another constraint, which says that if two actions in PP can be parallel, they are either:

- linked to different vessel

  or

- linked on the same vessel but one of them is *refuel* action and the other is either *load* or *unload* action

Also this constraint is only a necessary condition, for a pair of actions to be parallel, too. But conjunction of all of the conditions gives us a sufficient condition for any pair of actions in Petrobras domain to be parallel. Let's call a pair of actions that meet this condition *non-mutex actions*, and *mutex* if they do not.

[1]Mutually exclusive

Algorithm

```
   // starting time is 0
 1 currentTime = 0;
   // take first unprocessed action and set its start time
 2 currentAction = unprocessedActions[0];
 3 currentAction.setStartTime(currentTime);
 4 while currentAction != null do
       // add all parallel actions to the queue
 5     enqueueAllParallelActions();
       // obtain a new world state and set a new current time
 6     applyAction();
 7     currentTime = currentAction.getEndTime();
 8     if !actionQueue.empty() then
           // take next parallel action
 9         currentAction = actionQueue.poll();
10     else if !unprocessedAction.empty() then
           // take next unprocessed action and set its start time
11         currentAction = unprocessedActions.poll();
           currentAction.setStartTime(currentTime);
12     else
           // set current action to null to end the loop
13         curentAction = null;
14     end
15 end
```

Algorithm 1: Scheduling - pseudocode of the main method

```
 1 forall the unprocessedActions do
 2     if a vessel used in action was not tried yet then
 3         if action is not mutex with currentAction and all of the enqueued
           actions then
               // add non-mutex actions to the queue
 4             actionsQueue.put(action);
 5             action.setStartTime(currentTime);
 6         else
               // do not try this vessel in current iteration again
 7             remember vessel already tried
 8         end
 9     end
10 end
```

Algorithm 2: Scheduling - pseudocode of the enqueueAllParallelActions method

In Java generator method, I have used a *PriorityQueue<Action>* data structure (called *actionsQueue*), which stores objects (of the *Action* type) in a queue, sorted by their *starting time*. In the step 1, the first action is taken from the list of unprocessed actions (the list of actions parsed from SGPlan's output in the original order). The start time of the first action is 0.

We iterate through unprocessed actions. In every iteration all the actions that are non-mutex with current action are removed from the list of unprocessed actions and inserted into *actionsQueue*. When moving an action from *unprocessedActions*[2] to *actionsQueue*, that action needs to be non-mutex with every action that is already in *actionsQueue*. By postponing actions we ensure that the constraints on resources are not broken (for example, by waiting for resources to become available). The only resource, which is constrained in Petrobras domain is the capacity of the docks and the platforms. The second situation - where action c is executed between actions a and b, action a provides precondition p for action b and action c destroys p - is called *a thread*. In this case we postpone action a after the termination of action c. When this is done, *currentAction* is applied. This means that its effects are applied and thus a new world state is obtained. In the end of an iteration, we need to determine the next action for the next iteration. This is pretty simple - take the first action from *actionsQueue* (they are sorted by start time) and if it is empty, take the first action from *unprocessedActions* list. If that is empty too, return null as next action, so the main loop is over.

4.3 Example

4.3.1 Gantt Viewer introduction

Gantt Viewer 1.0[Ska09] is used for viewing a schedule. Petrobras schedule generator generates a plan in XML format, defined by Gantt Viewer, so our results can be displayed and reviewed easily.
Gantt Viewer has two basic views - *Tasks* and *Resources*. The task view is simply a Gantt chart[Fou12b]. Every line in a task view represents a single action - including a name, a start time, a finish time and a duration of the selected action on the left. On the right, there is a rectangle drawn on a time line. A schedule which contains many actions could be very long and unclear. On the other hand, the Resources view groups actions with the same "resource" into a single line. A resource is any object linked to an action. For the best and for the most detailed view of schedule, all parameters of an action are used as a resource. For example, action (LOAD V1 P1 C2) has three resources - vessel V1, port P1 and cargo item P2. That gives us a possibility to see which vessel executes which actions. Moreover, we are able to look at actions linked to each cargo item and each location very easily. If two or more actions are occurring at the same time at the same resource, Gantt Viewer displays both actions parallel to each other.

[2] An ArrayList<Action> where unprocessed actions are stored

4.3.2 Three vessels example

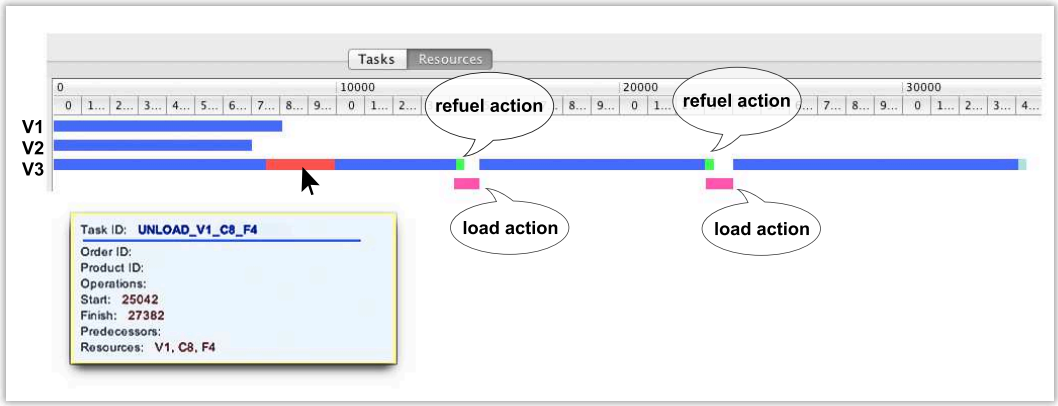


Figure 4.1: Schedule example

Figure 4.1 shows an example of a schedule of the *nice problem*[3] with three vessels (V1, V2, V3). Each blue line is actually a sequence of rectangles, each representing single action. As long as there is no gap between actions (when one finishes, another starts immediately), there is also no gap between rectangles. However, when a user hovers a mouse above the line, Gantt Viewer changes a color of the action's rectangle and displays all the details about given action. In figure 4.1, Gantt Viewer shows details about unload action.
We can see that vessels performs actions independently. In this example, none of the vessels needs to wait for other vessel to perform some action in order to continue. Moreover, we can see that vessel V3 refuels during loading and that this pair of actions runs parallel. After loading is done, a vessel starts to execute another action. More examples of schedules shown in Gantt Viewer, more screenshots and all features described can be found in section 5.2.6 - Actual schedules examples.

4.4 Computing costs

Not too difficult but very important part is to compute costs of a schedule. It is done during the post-processing after a schedule is generated.

- The most important one, *total fuel* consumed by vessels, is used as a metric for planners. To compute it, we need to remember a list of cargo items loaded aboard for each vessel, because it influences fuel consumption of a vessel. Therefore, for every *load* action, cargo item is added to the list of vessels and for every *unload* action it is removed.
- The *makespan* is simply the finishing time of the latest executing action.
- Every action is taken by one of the vessels. We keep the list of *vessels used* in the schedule using Java HashSet.

[3]Nice problem was described in section 3.3.3

- *Docking cost* is linearly dependent on time between docking and undocking a vessel. The price is $1000 an hour. Thus, we keep a track of docking start time for every vessel and compute the cost when a vessel undocks.

4.5 Schedule generator and cost computing tool in Java

We have already mentioned that schedule generating and cost computing algorithms are implemented in Java. Similarly to *ProblemGenerator*, the program developed to generate problem instances in PDDL - *ScheduleGenerator* generates schedules using a problem instance and the corresponding sequential plan. Again, it is wrapped up with shell script called *generate_schedule* and has following parameters and options.
Required paramenters:

1. path to a problem defined in PDDL file
2. path to a sequential plan in SGPlan output format

By running *generate_schedule* with only required parameters, *ScheduleGenerator* computes a schedule and displays plan metrics (costs) in human-readable format. The example of such output with metrics of the *nice problem*:

```
1 Total fuel              2612
2 Total vessels used      3
3 Makespan                572.73
4 Docking cost            354000.0
```

Optional parameters (modifying output):

-csv Outputs schedule costs as values separated by semicolons (CSV standard, separator: ';' and decimal separator ',') instead of human-readable format. Useful for batch computing and viewing in MS Excel, OpenOffice etc. The format of output is `Fuel;Vessels;MakeSpan;Docking Cost`.

-i outputs a schedule in XML format, compatible with Gantt Viewer ("i" comes from iGantt, another name of the viewer)

5. Experiments and results

Two planners have been tried during the work - SGPlan[HW06] and Probe[LG11]. I have found out that Probe ignores all the *fluents* in PDDL files, without any warning. It would be very useful and time saving if *:fluents* requirement was rejected right from the beginning. Ignoring fluents makes Probe planner useless for our purpose, therefore we will focus only on the results of SGPlan.
In this chapter we are going to discus mainly the performance and the results of SGPlan, but also how the whole model was benchmarked.

5.1 The design of experiments

5.1.1 Background

Both generators (*ProblemGenerator* and *ScheduleGenerator*) were designed as command-line Java applications, wrapped by shell scripts (*generate_pddl* and *generate_schedule*) and using text as input and output. This gives great conditions for batch unix/linux testing and processing. Moreover, *ScheduleGenerator* has option for using CSV (comma separated values) format as output.
The experiments ran on 64bit Ubuntu linux OS, Intel Core i7 SandyBridge 3.4GHz processor PC.

5.1.2 The design

Basically, we need to find out, how each of three of variables - cargo count, fuel capacity and vessels count - influence the costs - total fuel, number of vessels used, makespan, docking cost - and also processing time and success rate. We have designed 4 groups of experiments. Each group has the same vessel count and fuel capacity. The number of cargo items is from 1 to 16 in each group.

- *Group A* with 10 vessels and fuel tank capacity of 600 liters.
- *Group B* with 10 vessels and fuel tank capacity of 800 liters.
- *Group C* with 10 vessels and fuel tank capacity of 1000 liters.
- *Group D* with 3 vessels and fuel tank capacity of 600 liters.

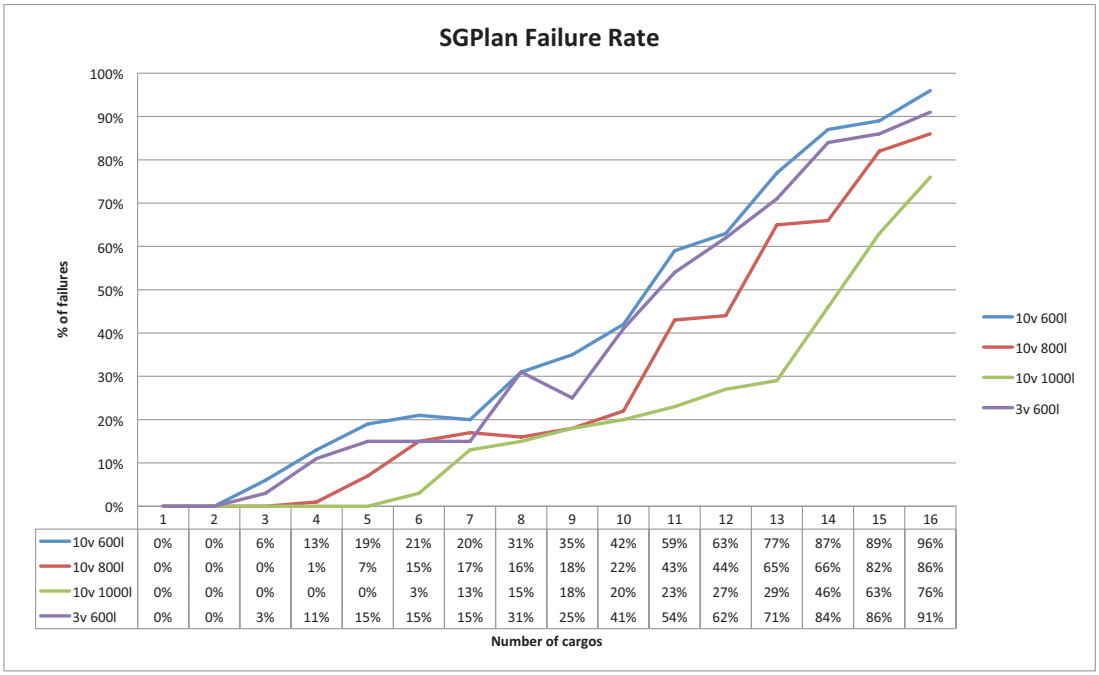
We need the results to be accurate, so we iterated every experiment 100 times. Every time with a different seed for the *ProblemGenerator*. Then we calculate the average value of all runs for which a solution was found by a planner.
The total count of runs is thus: 4 groups * 16 experiments * 100 runs = 6400. Assuming that it takes 2 minutes (on average) to calculate every run, it takes approximately two and a half day to calculate the results using all 4 threads on multicore i7 processor.

5.2 Results

The important fact is, that *SGPlan* computing times used in charts are average values calculated ONLY from *runs* that were successful - only when SGPlan returns a valid plan. So, however the difference between computing times in chart is very small, it took SGPlan about 45 minutes to exit when it did not find a solution. For example, the experiment (100 runs) with 10 vessels, 600 liters of fuel and 15 cargo items took *SGPlan* about 60 hours.

5.2.1 Computing time and success/failure rates

SGPlan is not a complete algorithm. The question is, how many cargo items in our groups[1] can SGPlan handle. Following graphs show how cargo items count influences failure rate and computing time.

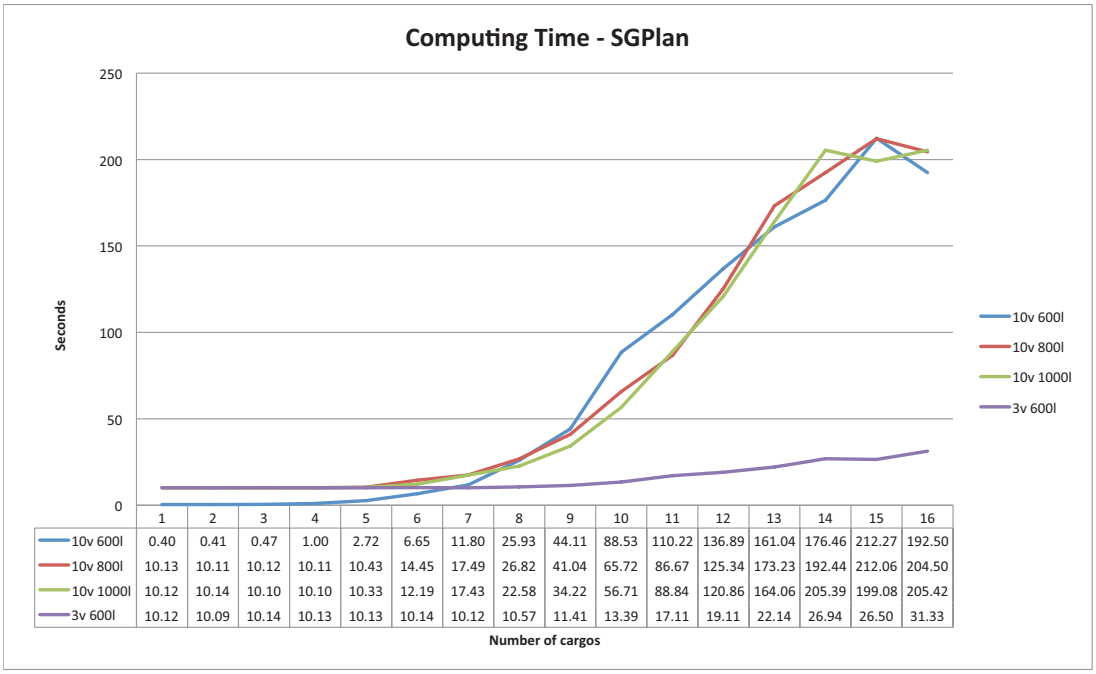


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
10v 600l	0%	0%	6%	13%	19%	21%	20%	31%	35%	42%	59%	63%	77%	87%	89%	96%
10v 800l	0%	0%	0%	1%	7%	15%	17%	16%	18%	22%	43%	44%	65%	66%	82%	86%
10v 1000l	0%	0%	0%	0%	0%	3%	13%	15%	18%	20%	23%	27%	29%	46%	63%	76%
3v 600l	0%	0%	3%	11%	15%	15%	15%	31%	25%	41%	54%	62%	71%	84%	86%	91%

Figure 5.1: Failure rate of SGPlan

Failure rate values do not differ a lot, however, failure rate is related to fuel capacity of vessels. The bigger the fuel tank, the more successful SGPlan is. The fact is that even the smallest fuel capacity (600l) is enough for a vessel to travel between all of the locations and return to one of the refuel locations. It is clear that success rate of SGPlan is lower than 50% when the number of cargo items is more than 10.

[1] described in section 5.1.2



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
10v 600l	0.40	0.41	0.47	1.00	2.72	6.65	11.80	25.93	44.11	88.53	110.22	136.89	161.04	176.46	212.27	192.50
10v 800l	10.13	10.11	10.12	10.11	10.43	14.45	17.49	26.82	41.04	65.72	86.67	125.34	173.23	192.44	212.06	204.50
10v 1000l	10.12	10.14	10.10	10.10	10.33	12.19	17.43	22.58	34.22	56.71	88.84	120.86	164.06	205.39	199.08	205.42
3v 600l	10.12	10.09	10.14	10.13	10.13	10.14	10.12	10.57	11.41	13.39	17.11	19.11	22.14	26.94	26.50	31.33

Figure 5.2: Computing time of SGPlan

Computing time is quite similar in groups A,B,C and much shorter in group D. It is caused by number of vessels available in group D - SGPlan has much smaller search space to explore.

5.2.2 Vessels used count

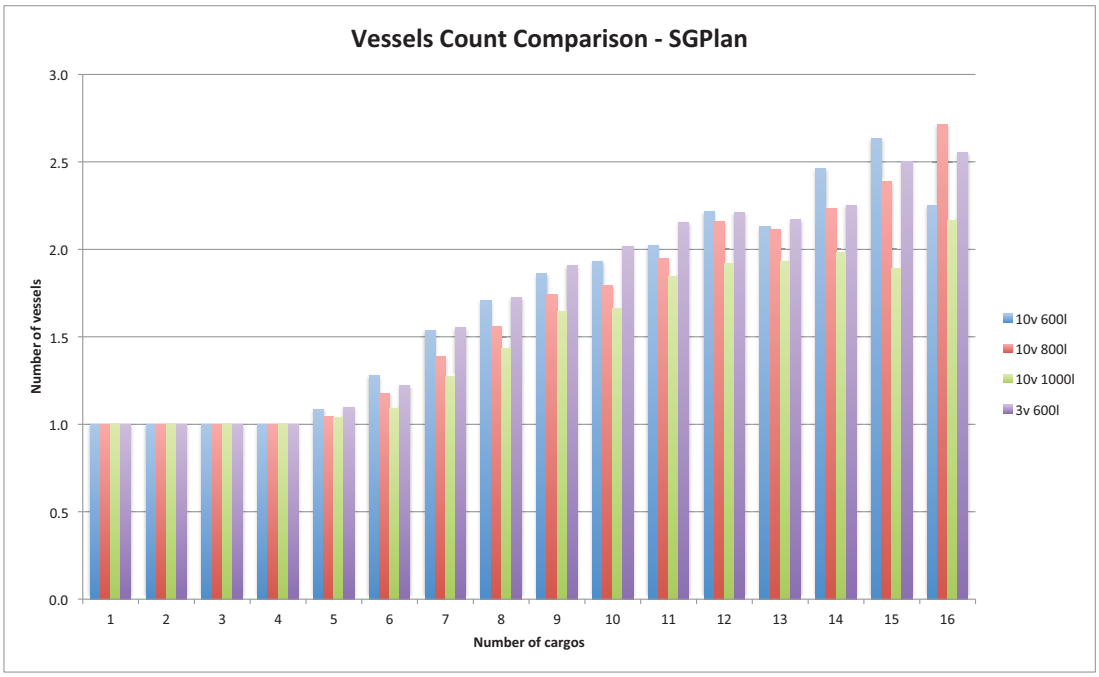
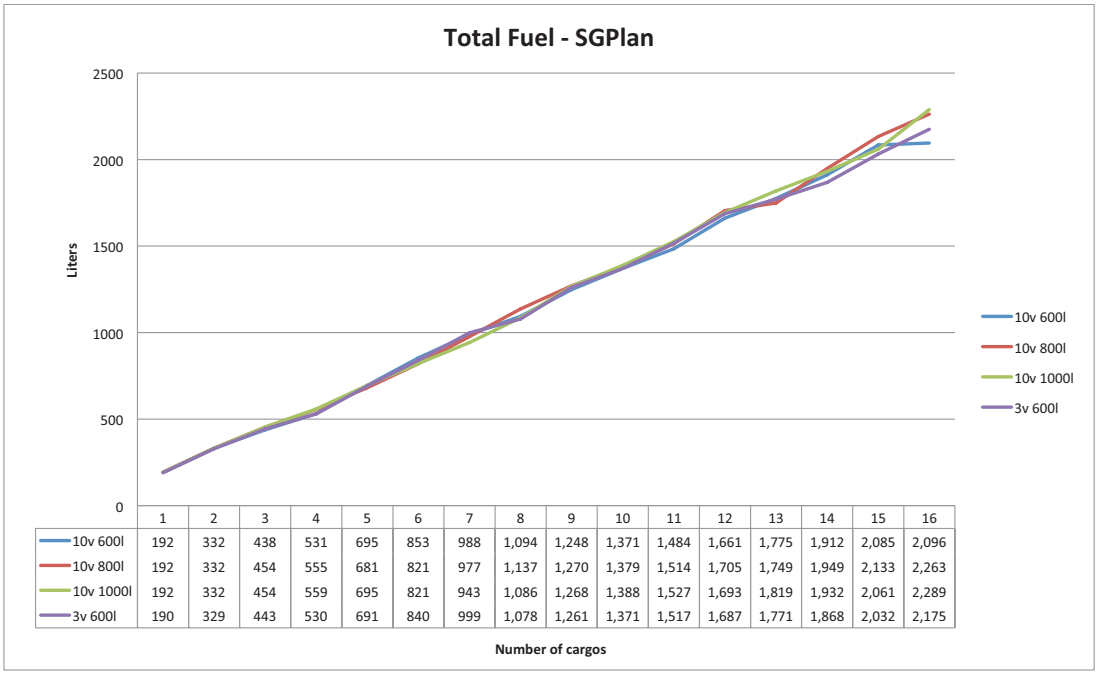


Figure 5.3: Vessels used count - SGPlan

SGPlan tends to choose a little more vessels as the number of cargo items increase. To optimize fuel cost, it chooses as few vessels as possible. On the other hand, the makespan of the schedules is thus very long. As we can see, the average value of vessels used is slightly smaller when the fuel capacity vessels is bigger.

5.2.3 Fuel consumption

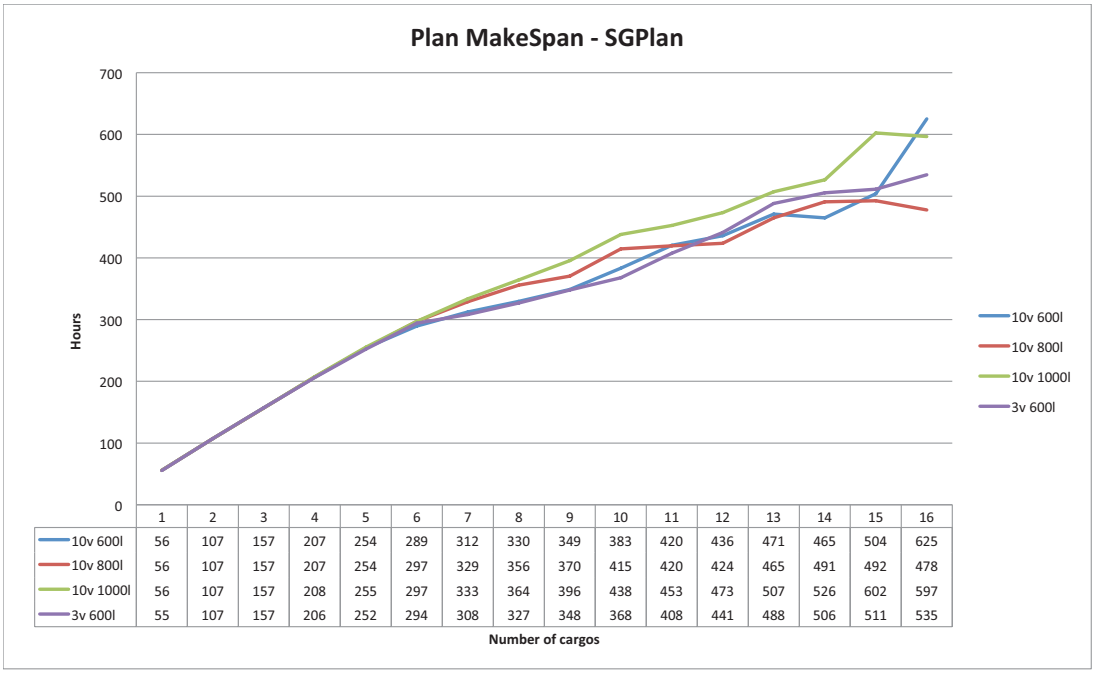


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
10v 600l	192	332	438	531	695	853	988	1,094	1,248	1,371	1,484	1,661	1,775	1,912	2,085	2,096
10v 800l	192	332	454	555	681	821	977	1,137	1,270	1,379	1,514	1,705	1,749	1,949	2,133	2,263
10v 1000l	192	332	454	559	695	821	943	1,086	1,268	1,388	1,527	1,693	1,819	1,932	2,061	2,289
3v 600l	190	329	443	530	691	840	999	1,078	1,261	1,371	1,517	1,687	1,771	1,868	2,032	2,175

Figure 5.4: Total fuel cost of SGPlan

Because SGPlan optimizes fuel cost only, it uses as few vessels as possible. It uses only one vessel most of the experiments. The total amount of fuel is therefore related to the number of cargo items.

5.2.4 Makespan



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
10v 600l	56	107	157	207	254	289	312	330	349	383	420	436	471	465	504	625
10v 800l	56	107	157	207	254	297	329	356	370	415	420	424	465	491	492	478
10v 1000l	56	107	157	208	255	297	333	364	396	438	453	473	507	526	602	597
3v 600l	55	107	157	206	252	294	308	327	348	368	408	441	488	506	511	535

Figure 5.5: Makespan of SGPlan

SGPlan does not optimize the makespan and uses one vessel almost every time. Therefore the makespan is influenced only by a number of cargo items we need to transport. Comparing different groups, the makespan values are also very similar.

5.2.5 Actual schedules examples

Example 1

Let us show an example of a single particular problem (*Example1*) with 1 cargo item, 10 vessels and 600 liters fuel capacity. Cargo item C1 is at port P2 and its destination platform is F3.

Following table shows metrics of schedule generated by SGPlan:

```
        Total fuel  201
Total vessels used  1
          Makespan  36.29
     Docking cost   13000.0
```

The sequential plan of actions:

```
(NAVIGATE-EMPTY-VESSEL V10 A2 P2)
(DOCK-VESSEL V10 P2)
(LOAD-CARGO V10 C1 P2)
(UNDOCK-VESSEL V10 P2)
(NAVIGATE-NONEMPTY-VESSEL V10 P2 F3)
(DOCK-VESSEL V10 F3)
(UNLOAD-CARGO V10 C1 F3)
(UNDOCK-VESSEL V10 F3)
(NAVIGATE-EMPTY-VESSEL V10 F3 A2)
```

Following figures show the schedule of Example1 problem explored in Gantt Viewer in *Tasks view* and in *Resources view*:

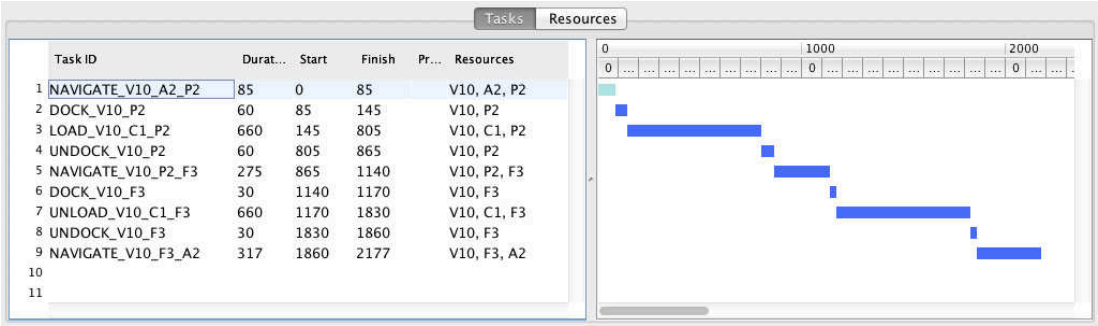


Figure 5.6: Schedule computed by SGPlan - problem Example1 - Tasks view

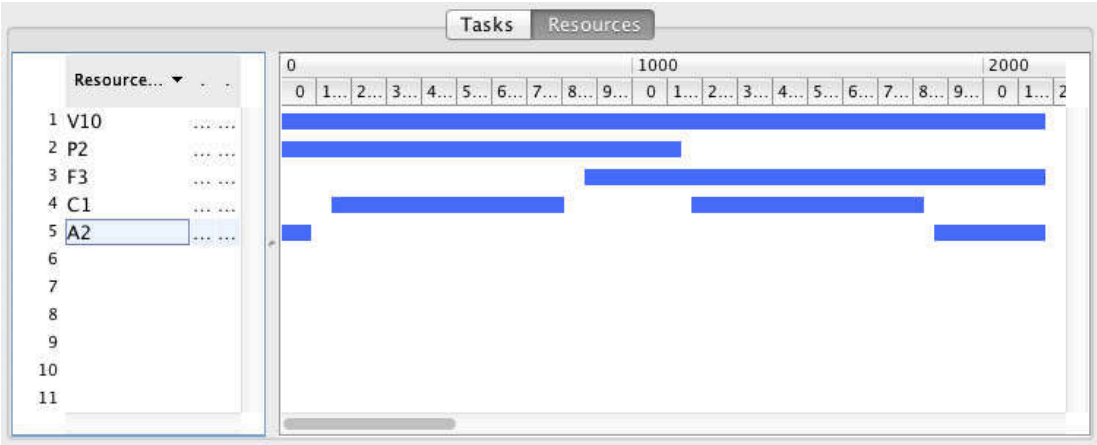


Figure 5.7: Schedule computed by SGPlan - problem Example1 - Resources view

Example 2

Another example (*Example2*) contains 13 cargo items and 3 vessels and again with 600 liters fuel tank.

The metrics of plans generated by SGPlan are:

```
         Total fuel  1893
Total vessels used  2
           Makespan  657.03
      Docking cost  379000.0
```

The schedule in Gantt Viewer's *Task view* is way too long to be shown here, but the *Resources view* clearly shows how two vessels V2 and V3 transport all 13 cargo items to their final destinations. The schedule for the vessel V3 is in the first row and the one for the vessel V2 is in the second row. It is important to say that all the actions in vessels' rows contain complete information about the whole schedule. It is because every action in a whole schedule is executed by exactly one of the vessels. Next rows contain redundant information - for every location and every cargo item (basically for every other object in a problem) a list of actions that are somehow linked to the object. It is quite useful: for each cargo item we are able to find out very quickly where, when and by which vessel it was loaded and unloaded.

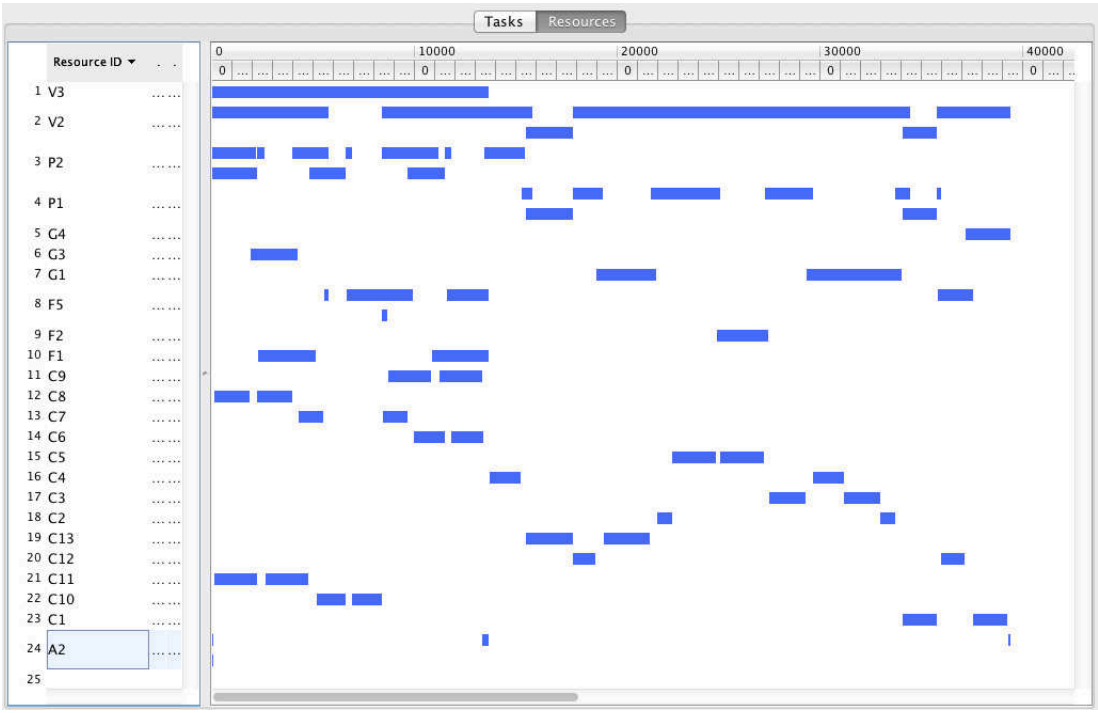


Figure 5.8: Schedule computed by SGPlan - problem Example2

5.3 Comparison to other approaches

As it was said before, this problem has been one of the domains of the Challenge Track of the ICKEPS 2012[2], as a part of the ICAPS 2012 conference. Therefore, a team composed of Roman Barták, Daniel Toropila, Filip Dvořák, Otakar Trunda and Martin Hanes was gathered to come up with a number of approaches and compare their results. Finally, these three approaches were implemented, tested and used for a number of problems in order to bring a valuable contribution to the ICAPS 2012 conference. After participating on ICKEPS 2012, the obtained results were also submitted to the ICTAI 2012 conference [Tor+12]. Along with this approach - using a classical planning to find sequential plan and make a parallel schedule from it - another two approaches were implemented:

- Temporal planning using Filuta Planning System.
- Ad-Hoc approach using Monte-Carlo Tree Search.

5.3.1 Temporal planning and Filuta description

This approach extends Petrobras PDDL domain definition by adding explicit durations to actions and additional concurrency conditions.
Filuta [DB10] is a planning system that operates with explicit time and models resources contained in the problem individually, using efficient solving techniques dependent on the real-world behavior of the modeled resource. The planner models the state of theworld using the *SAS+ representation* and adds the temporal annotations of the world state using a *temporal database* [NGT04] for each state variable.
The main optimization criterion of the planner is the makespan, however the planner sacrifices completeness and optimality in favor of the performance. The key idea for the performance boost is the division of the planning problem into sub-problems, where each of them contains only a single goal, resembling thus the original STRIPS algorithm for classical planning[Tor+12].

5.3.2 Ad-Hoc planner

Ad-Hoc planner was implemented "from-the-scratch" in contrast with the other two approaches, where we use existing methods (systems, planners). It was decided to use MonteCarlo Tree Search algorithm, because it works with an evaluation function and uses it during the search[Cha+08]. The function used is:

$$f(\pi) = usedFuel + 10 \times countOfActions + 5 \times makespan$$

5.3.3 Experiments description and results

To compare the performance and metrics (costs) of all three approaches, four groups of problem instances were designed:

- *Group A* with 3 vessels and fuel tank capacity of 600 liters.

[2]International Competition on Knowledge Engineering for Planning and Scheduling

- *Group B* with 10 vessels and fuel tank capacity of 600 liters.
- *Group C* with 10 vessels and fuel tank capacity of 800 liters.
- *Group D* with 10 vessels and fuel tank capacity of 1000 liters.

For each group, 15 problem instances (each having 1 to 15 cargo items) were generated. Moreover, another problem instance called "The sample problem" was designed. Its properties were set to the exact values from the Petrobras domain and problem specification[Igr12], where this problem was described as an example. Altogether, all the three planners were attempting to solve 61 problem instances. Finally, we compared their metrics - fuel used, vessels used, makespan and docking cost.

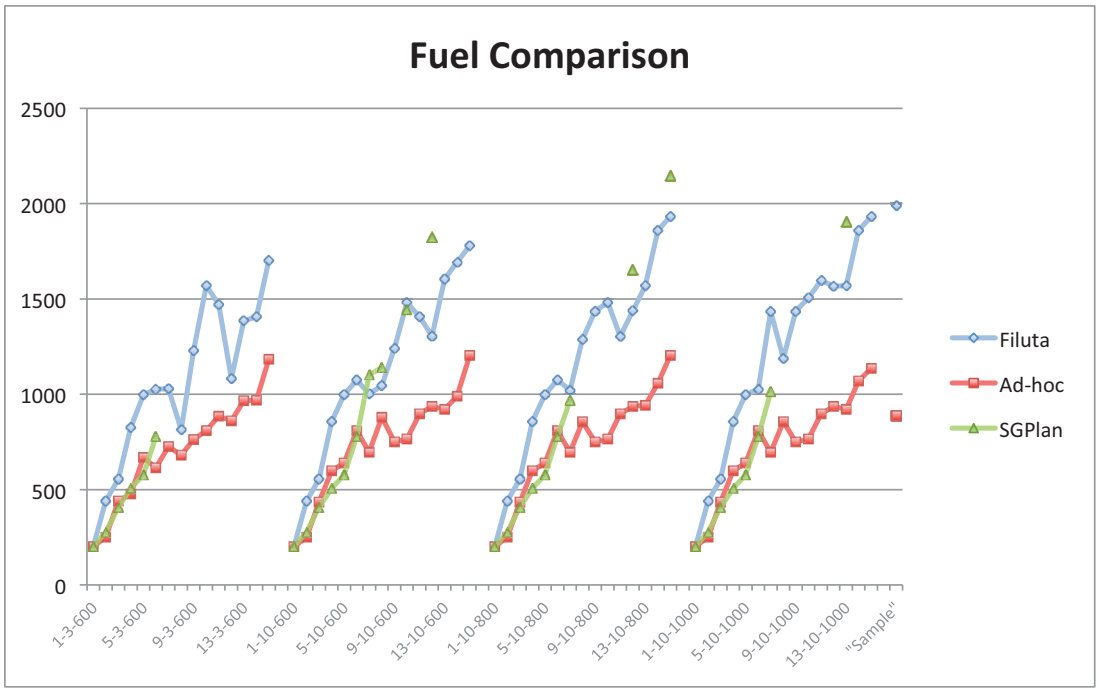


Figure 5.9: Approaches comparison - fuel consumption

As Filuta is optimizing the makespan only, the fuel consumption is the worse of all three approaches. SGPlan and Ad-Hoc do not differ a lot for small problem instances, but it is clear from the gaps in SGPlan's series that the success-rate is relatively small compared to Ad-Hoc for bigger problems.

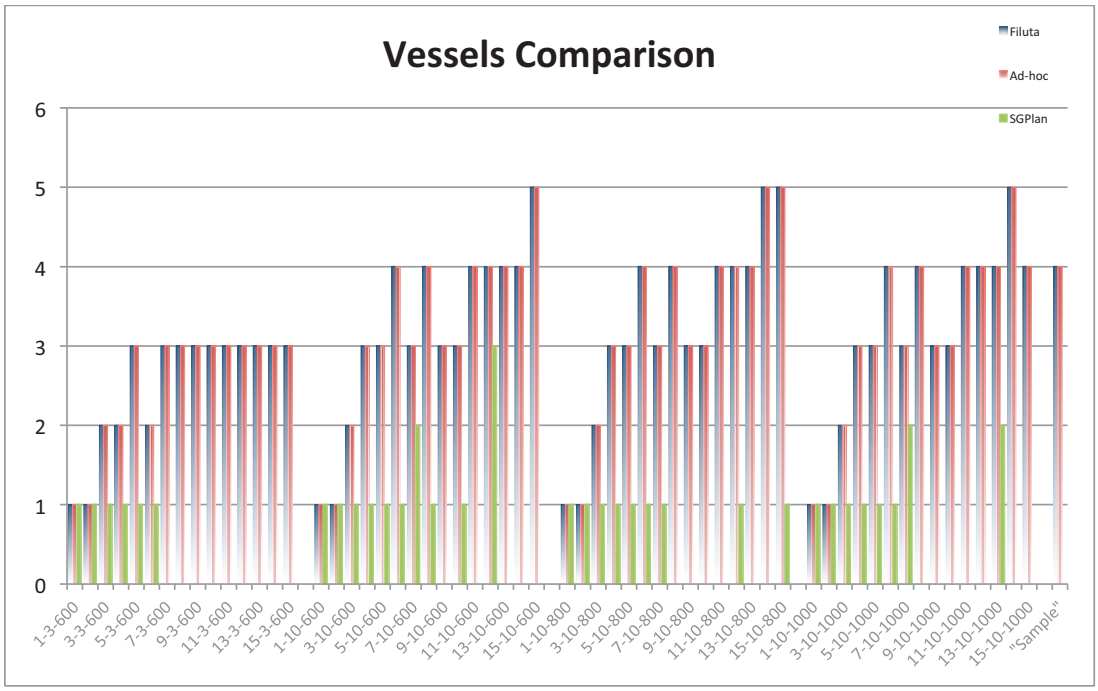


Figure 5.10: Approaches comparison - vessels count

SGPlan was using mostly only one vessel. Filuta and Ad-Hoc approaches were using similar number of vessels - number of vessels is limited by port and platform capacity, because loading/unloading actions take majority of the whole schedule time.

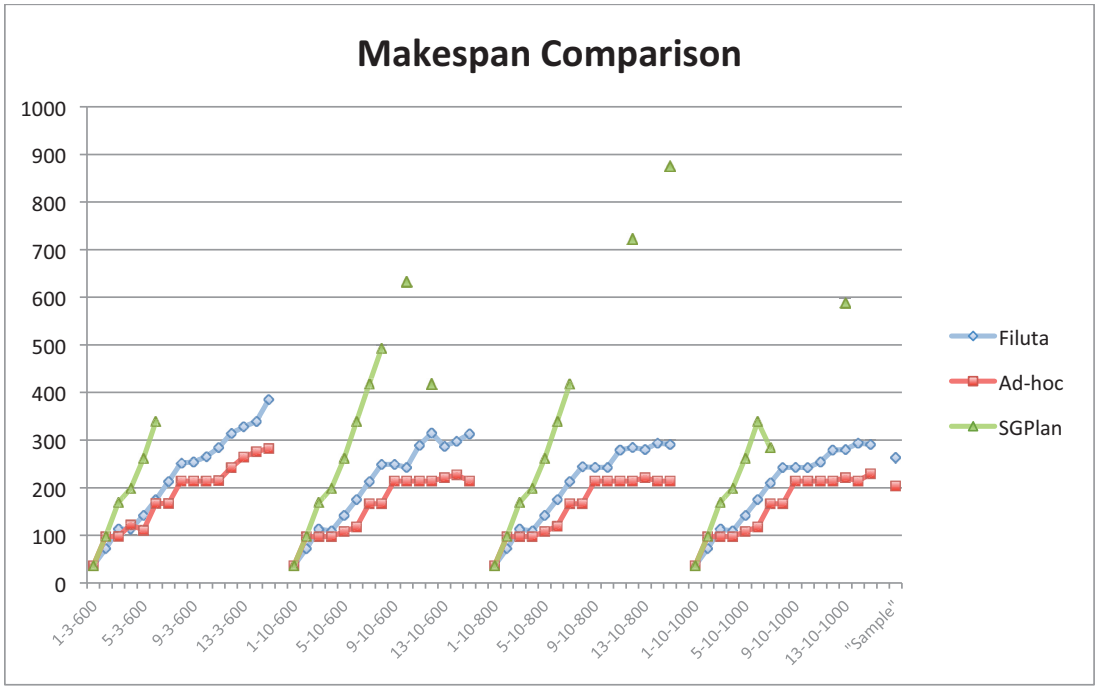


Figure 5.11: Approaches comparison - makespan

The makespan of SGPlan's schedules is clearly the worst. It comes from the fact that it uses mostly only one vessel. Again, the domain-dependent solver (Ad-Hoc) has the best results.

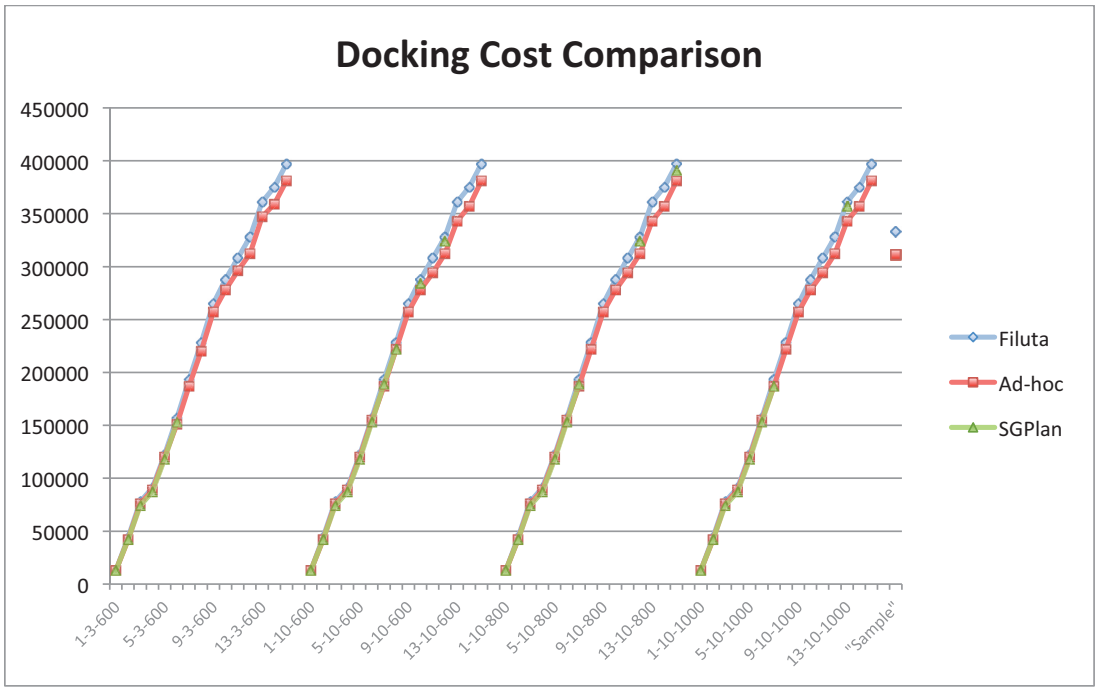


Figure 5.12: Approaches comparison - docking cost

Docking and undocking actions take overwhelming minority of a whole schedule, so they have minimal impact on the docking cost. On the other hand, load and unload actions mostly influence docking cost and they both are directly related to the number of cargo items. Therefore the docking cost directly correlates with the number of cargo items (sum of weights of all the items).

In conclusion, the Ad-Hoc approach achieved the best results in overall. The results of SGPlan were not very bad, but it worked for small problem instances only. The disadvantage of Filuta approach was the inability to optimize fuel cost metric.

5.3.4 More information from ICKEPS 2012

The article *Planning and Scheduling Ship Operations on Petroleum Ports and Platforms* by Tiago Stegun Vaquero and his team describes how their group was solving Petrobras challenge track including modeling PDDL and planning results. In their conclusion they wrote:

Experimental results showed that in both cases some planners can provide valid solutions for the problem, however, they struggle to provide solutions to more realistic problems. It is important to note that few planners can deal with such a combination of PDDL features. Therefore, the resulting PDDL model brings interesting challenges even for the state-of-the-art planners.[Vaq+12]

We absolutely agree with this statement as we had similar experience during the work. Independently of our team they have tried following three planners: Metric-FF, SGPlan6 and MIPS-xxl 2008, and found that SGPlan performs the best in most cases.

6. Graphical User Interface

6.1 Introduction

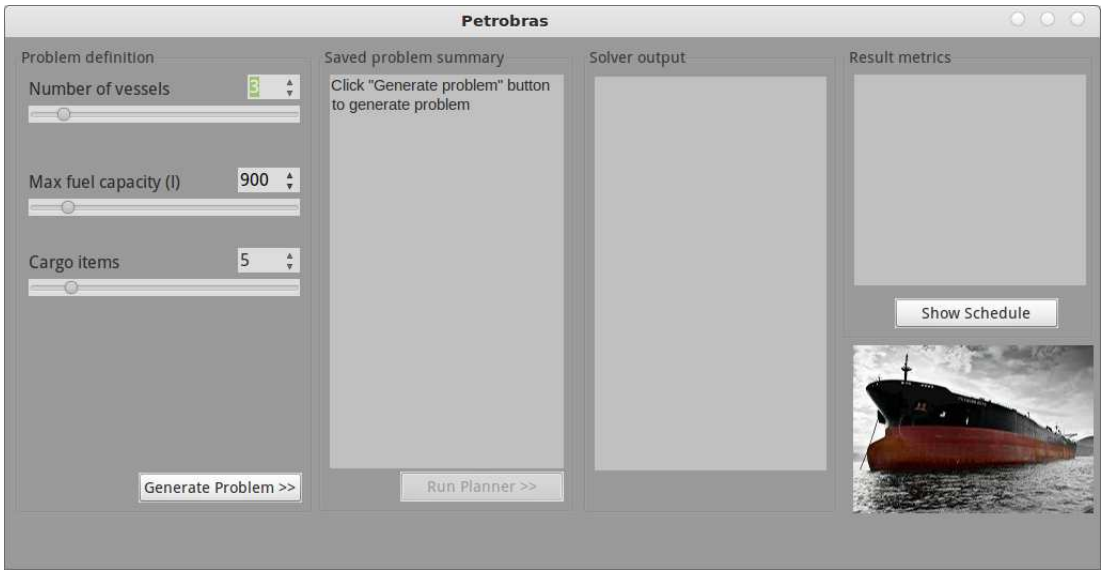


Figure 6.1: Petrobras GUI - Main window

A simple GUI was created in order to be able to find out, how SPGlan processes Petrobras problems. The layout is divided to 4 sections:

- Problem definition
- Saved problem summary
- Solver output
- Result metrics

Following subchapters provide the detailed description of these sections.

6.2 Problem definition section

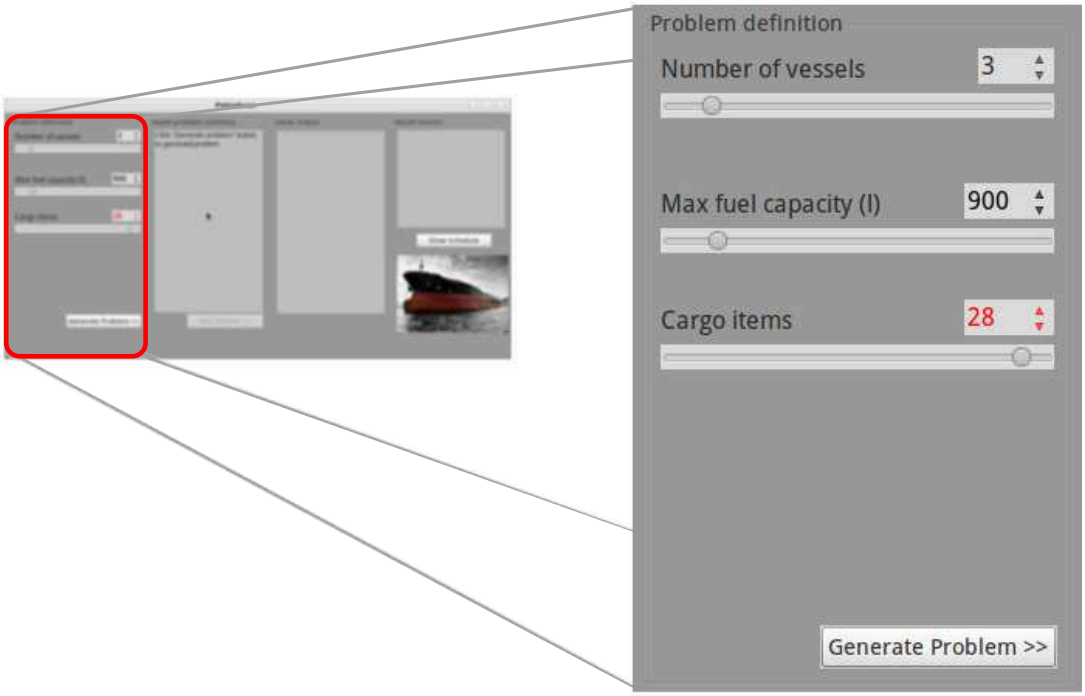


Figure 6.2: Petrobras GUI - Problem definition section

In *Problem definition section* we change variables of the model using sliders and numeric spinners. When a value is out of "safe" range - the range when SGPlan tends to return a solution within several seconds - the color of the number changes to red. For example, the number of cargo items in the previous figure is out of safe range.

By clicking *Generate Problem* button, *generate_pddl* script is called with corresponding options and a problem PDDL file is generated.

6.3 Saved problem summary section

This section briefly informs about the problem instance we generated in the previous step. After *Generate Problem* button is pressed and PDDL problem file is generated, the TextBox in *Saved problem summary* section fills up with summary information about the problem so we are able to double-check if the values are correct. Also the *Run Planner* button is activated and we are able to run SGPlan to compute the plan by pressing that button.

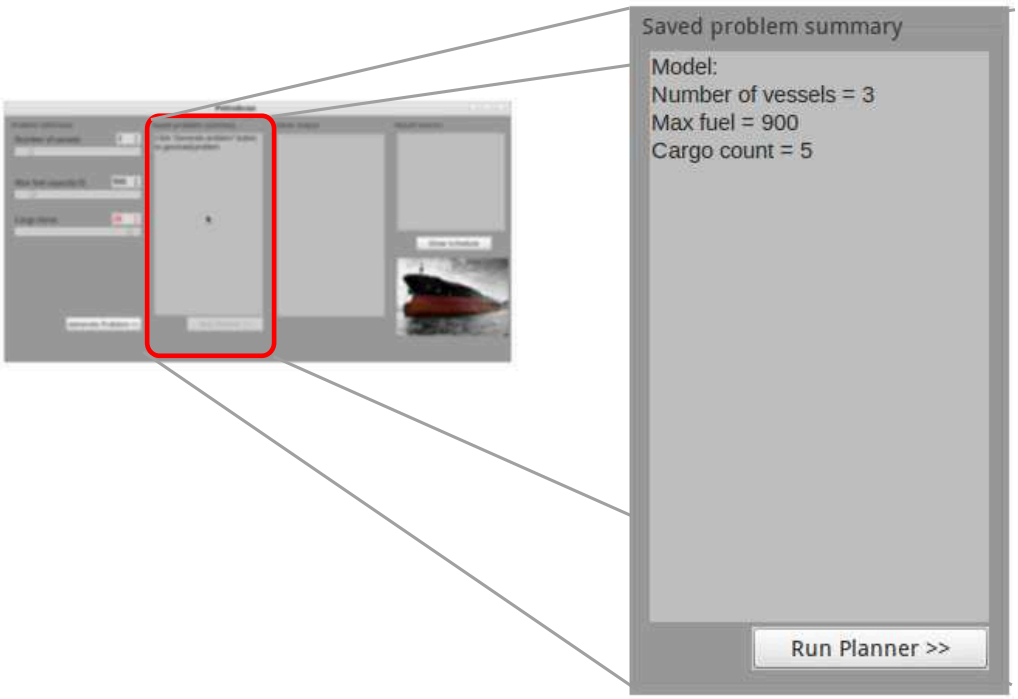


Figure 6.3: Petrobras GUI - Saved problem summary

6.4 Solver output section

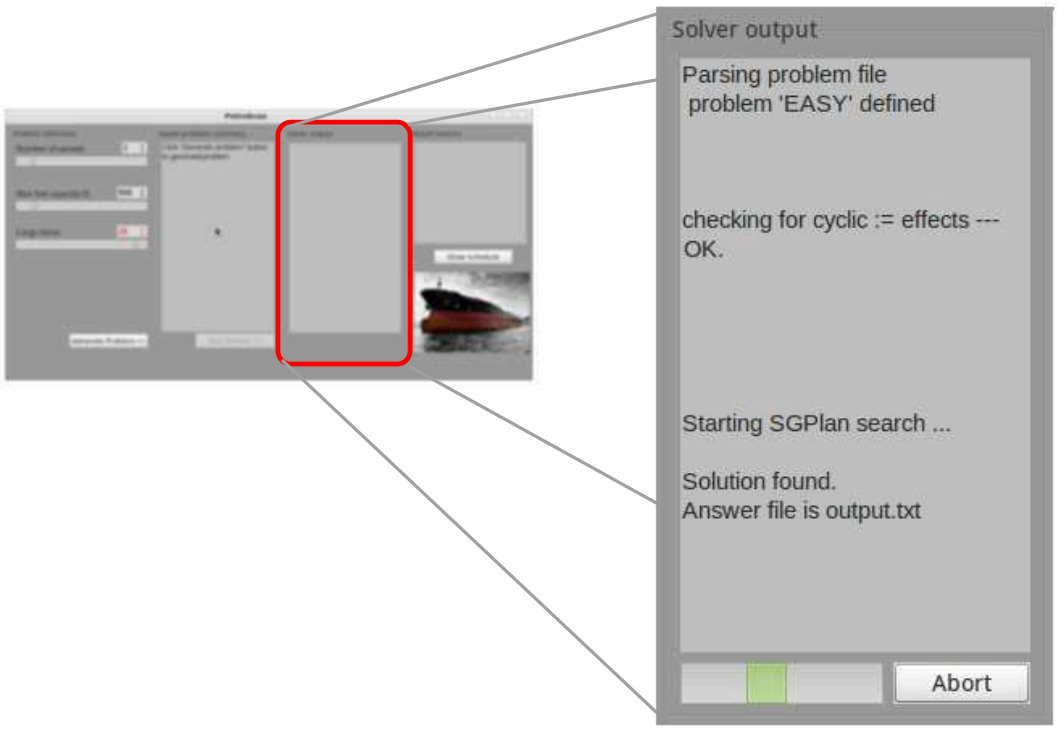


Figure 6.4: Petrobras GUI - Solver output

Figure 6.4 shows standard SGPlan's output as it is processing PDDL files and searching the solution. Along with this output the indefinite progress bar and Abort button appears. The Abort button simply kills all SGPlan's processes and interrupts the search for solution.

6.5 Result metrics section

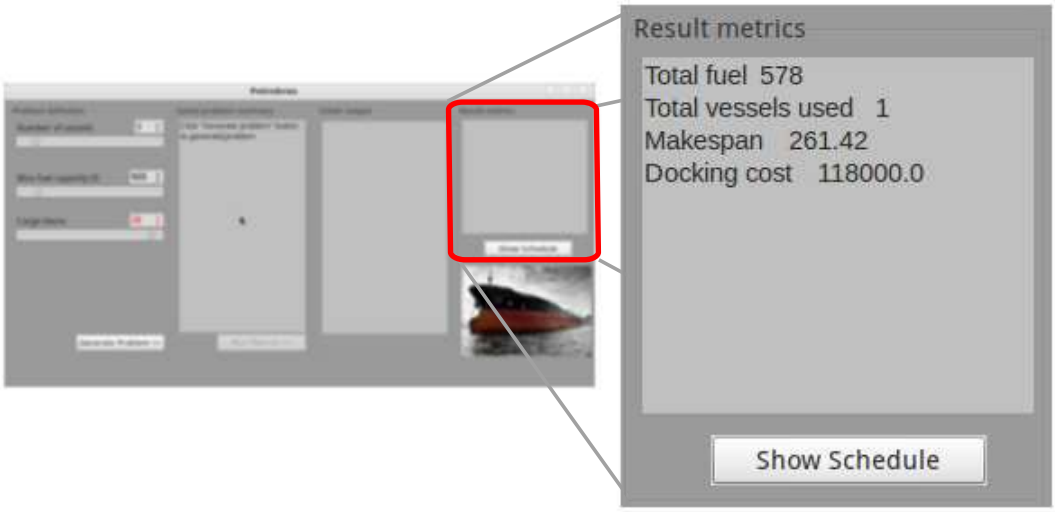


Figure 6.5: Petrobras GUI - Results metrics

After SGPlan successfully exits and returns a sequential plan, *generate_schedule* script is run and a parallel schedule is generated. Along with that, the metrics (costs) of that schedule are computed and shown in *Results metrics* section as shown on Figure 6.5.
The *Show Schedule* button opens generated schedule in Gantt Viewer, which was described earlier in the text.

6.6 Notes about the GUI

- Right after clicking *Run SGPlan* button, the button is disabled and it is enabled again after user generates a new problem instance. It comes from the fact that SGPlan is generating its results deterministicaly. In other words, running SGPlan on the same problem generates the same result and thus there is no reason to run SGPlan more than once before another problem instance is generated (by user).
- If user generates a new problem instance and clicks *Runs SGPlan* button while SGPlan is still running on a previous instance, the older instance is killed.
- *Show Schedule* button always opens a new Gantt Viewer window without closing the previous ones.

- The GUI is implemented in Java and designed in *SWT: The Standard Widget Toolkit*[Fou12a].
- SGPlan is designed to be used on a Linux platform. Because it uses gigabytes of RAM memory, the GUI and the whole *Petrobras_GUI* package is designed and tested to be used on 64bit Linux (with standard GTK libraries) only.

Conclusion

We explored the Petrobras problem in this work, introduced planning and scheduling and its existing tools used by the planning community. We discussed PDDL language more in detail describing its evolution and features and wrote about how the Petrobras domain was modeled in that language. We successfully modeled the domain in PDDL language and tried a couple of planners but only SGPlan (6^{th} version) was able to provide us valid results - sequential plans.

In chapter 4 we deeply analyzed scheduling a sequential plan into parallel one by explaining the theory and outlining the pseudocode of actual algorithm used. The next chapter - chapter 5 - examined the results and explained their characteristics. We optimized the *fuel-cost* metric only and found out, that *makespan* is not optimized indirectly as it is not correlated with fuel-cost metric. The makespan of the schedules was long because SGPlan returned plans which were using as few as possible vessels.

We examined the success rate of SGPlan, how long it calculates the result and all the metrics - total fuel used, makespan, cargo count and docking cost. We compared our results to the other approaches - Filuta and Ad-Hoc and explained the differences. The SGPlans results were also compared to the other team which implemented a similar approach - to compute a sequential plan and schedule the actions to create parallel plan. They found out the same result - using sequential planner and transforming a sequential plan into parallel schedule we are able to get satisfying results for small problem instances only. For bigger problem instances sequential planners do not return a valid solution. Also the quality of the schedules (in general, according to the metrics) we were able to get was not better than the Ad-Hoc planner which was optimizing a function combining of all the metrics.

We gave a quick guide to the GUI which was developed along with modeling for debugging purposes. It uses generators and tools and creates a graphical and user friendly interface and enables us to examine the results and display them in a very clear, user friendly way.

The future work could be to find a way to optimize more than one metric - for example by creating such optimization criterion that would include more vessels in a plan and thus improving the makespan indirectly.

Bibliography

[Cha+08] Guillaume Chaslot et al. "Monte-Carlo Tree Search: A New Framework for Game AI." In: *AIIDE*. 2008.

[DB10] Filip Dvorak and Roman Bartak. "Integrating Time and Resources into Planning." In: *Proceedings of the 2010 22nd IEEE International Conference on Tools with Artificial Intelligence - Volume 02*. ICTAI '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 71–78. ISBN: 978-0-7695-4263-8. DOI: 10.1109/ICTAI.2010.86. URL: http://dx.doi.org/10.1109/ICTAI.2010.86.

[Fou12a] The Eclipse Foundation. *SWT: The Standard Widget Toolkit*. 2012. URL: http://www.eclipse.org/swt/.

[Fou12b] Wikimedia Foundation. *Gantt chart*. July 4, 2012. URL: http://en.wikipedia.org/wiki/Gantt_chart.

[Fou12c] Wikimedia Foundation. *Planning Domain Definition Language*. July 2012. URL: http://en.wikipedia.org/wiki/Planning_Domain_Definition_Language.

[Has11] Patrik Haslum. *PddlExtension*. Sept. 2011. URL: http://ipc.informatik.uni-freiburg.de/PddlExtension.

[Hel11] Malte Helmert. *PDDL papers*. Apr. 2011. URL: http://ipc.informatik.uni-freiburg.de/PddlResources.

[HW06] Chih-Wei Hsu and Benjamin W. Wah. "The SGPlan Planning System in IPC-6." In: 2006.

[Igr12] Tonidandel Igreja Silva. In: (Jan. 13, 2012). URL: http://icaps12.poli.usp.br/icaps12/sites/default/files/ickeps/petrobrasdomain/ICKEPS%20Petrobras%20Domain%20v1.0.pdf.

[LG11] Nir Lipovetzky and Hector Geffner. "Searching with Probes: The Classical Planner PROBE." In: *International Planning Competition (IPC-7)*. June 2011.

[NGT04] Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning: Theory & Practice*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004, p. 1. ISBN: 1558608567.

[Ska09] Tomáš Skalický. *Interactive Gantt Viewer with Automated Schedule Repair*. 2009. URL: http://clp.mff.cuni.cz/iGantt.html.

[sta09] Latin business chronicle staff. *Petrobras Largest Company in Latin America*. June 2009. URL: http://www.latinbusinesschronicle.com/app/article.aspx?id=3440.

[Tor+12] Daniel Toropila et al. "Three Approaches to Solve Petrobras Domain." In: *Submitted to ICTAI 2012*. June 2012.

[Vaq+12] Tiago Stegun Vaquero et al. "Planning and Scheduling Ship Operations on Petroleum Ports and Platforms." In: *In proceedings of Scheduling and Planning Applications workshop (SPARK) 2012*. June 26, 2012.

List of figures

List of abbreviations

IPC	International Planning Competition
PDDL	planning domain definition language
PP	Petrobras problem
P&S	Planning and Scheduling
WA	Waiting area

Appendix A

Contents of the CD

Directory	Contents
DOC	Javadoc with code
EXECUTABLES	Compiled code in .jar files. GUI is started as java .jar application: `java -jar Petrobras_GUI.jar`
SOURCE	Source files - ProblemGenerator, ScheduleGenerator, Petrobras (GUI) directories with Eclipse projects and petrobras-domain.pddl text file with domain definition.
TEXT	This thesis in PDF

Appendix B

Petrobras domain PDDL

Listing 6.1: Petrobras domain PDDL

```
(define (domain petrobras)

    (:requirements :typing :action-costs :fluents)

    (:types
        location vessel cargo - object
         logistics_location waiting-area - location
         platform port - logistics_location
    )


    (:predicates

        ;; location is waiting area
        (is-waiting-area ?loc - location)

        ;; vessel is at location (platform, port, waiting area)
        (at ?vessel - vessel ?where - location)

        ;; vessel is docked in port or platform
        (is-docked ?v - vessel ?where - location)

        ;; platform is able to refuel a vessel
        (platform-can-refuel ?platform - platform)

        ;; cargo is at location
        (cargo-at ?c - cargo ?loc)

        ;; helping predicates
        ;; vessel was once docked at this location (navigate
            action negates this predicate)
        (vessel-once-docked-at-location ?v - vessel ?l - location
            )

        ;; vessel was once refueled at location (undock action
            negates this predicate)
        (vessel-once-refueled-at-location ?v - vessel ?l -
            logistics_location)
    )


    (:functions
        ;; the sum of fuel used by all the ships - to be
            minimized
        (total-fuel) - number

        ;; the free space in a vessel
        (vessel-free-capacity ?vessel - vessel) -  number

        ;; how many vessels can be docked in location
```

```
46          (free-docks ?loc - logistics_location)  - number
47
48          ;; max free capacity of a vessel
49          (max-vessel-free-capacity) - number
50
51          ;; current tank of a vessel
52          (fuel-level ?vessel - vessel) - number
53
54          ;; max fuel level (after refueling) independent on vessel
55          (max-fuel-level) - number
56
57          ;; the tank consumption - depending on the state of
                vessel (empty or not)
58          (navigation-cost-empty ?from ?to - location) - number
59          (navigation-cost-nonempty ?from ?to - location) - number
60          (navigation-cost-empty-to-nearest-refuel-loc ?
                waiting-area - location) - number
61
62          ;; the distance between locations
63          (distance ?from ?to - location) - number
64
65          ;; the weight of cargo
66          (cargo-weight ?c - cargo) - number
67      )
68
69      ;;navigate vessel if it's empty
70      (:action navigate-empty-vessel
71          :parameters (?vessel - vessel ?from ?to - location)
72          :precondition    (and
73                              (at ?vessel ?from)
74                              (= (vessel-free-capacity ?vessel) (
                                    max-vessel-free-capacity))
75                              (>= (fuel-level ?vessel) (
                                    navigation-cost-empty ?from ?to))
76                              (or
77                                  (not (is-waiting-area ?to))
78                                  (>= (-
79                                          (fuel-level ?vessel)  (
                                                navigation-cost-empty
                                                ?from ?to)
80                                      )
81                                      (navigation-cost-empty-to-
                                            nearest-refuel-loc ?to)
82                                  )
83                              )
84                              (not (is-docked ?vessel ?from))
85                          )
86          :effect (and
87                      (not (at ?vessel ?from))
88                      (at ?vessel ?to)
89                      (not (vessel-once-docked-at-location ?vessel
                            ?from))
90                      (decrease (fuel-level ?vessel) (
                            navigation-cost-empty ?from ?to))
91                      (increase (total-fuel) (navigation-cost-empty
                             ?from ?to))
92                  )
93      )
```

```
94
95
96      ;;navigate vessel if it's NOT empty
97      (:action navigate-nonempty-vessel
98          :parameters (?vessel - vessel ?from ?to - location)
99          :precondition    (and
100                             (at ?vessel ?from)
101                             (< (vessel-free-capacity ?vessel) (
                                    max-vessel-free-capacity))
102                             (>= (fuel-level ?vessel) (
                                    navigation-cost-nonempty ?from ?to
                                    ))
103                             (not (is-docked ?vessel ?from))
104                         )
105          :effect (and
106                      (not (at ?vessel ?from))
107                      (at ?vessel ?to)
108                      (not (vessel-once-docked-at-location ?vessel
                             ?from))
109                      (decrease (fuel-level ?vessel) (
                             navigation-cost-nonempty ?from ?to))
110                      (increase (total-fuel) (
                             navigation-cost-nonempty ?from ?to))
111                  )
112      )
113
114     ;;vessel loads cargo from location
115     (:action load-cargo
116         :parameters (?vessel - vessel ?c - cargo ?loc -
                logistics_location)
117         :precondition    (and
118                             ;; redundant but useful
119                             (at ?vessel ?loc)
120                             (is-docked ?vessel ?loc)
121                             (cargo-at ?c ?loc)
122                             (>= (vessel-free-capacity ?vessel) (
                                    cargo-weight ?c))
123                         )
124         :effect (and
125                     (not (cargo-at ?c ?loc))
126                     (cargo-at ?c ?vessel)
127                     (decrease (vessel-free-capacity ?vessel) (
                             cargo-weight ?c))
128                 )
129     )
130
131     ;;vessel unloads cargo in location
132     (:action unload-cargo
133         :parameters (?vessel - vessel ?c - cargo ?loc -
                logistics_location)
134         :precondition    (and
135                             ;; redundant but useful
136                             (at ?vessel ?loc)
137                             (is-docked ?vessel ?loc)
138                             (cargo-at ?c ?vessel)
139                         )
140         :effect (and
141                     (not (cargo-at ?c ?vessel))
```

```
                    (cargo-at ?c ?loc)
                    (increase (vessel-free-capacity ?vessel) (
                        cargo-weight ?c))
                )
    )

    ;; vessel refuels at refueling platform
    (:action refuel-vessel-platform
        :parameters (?vessel - vessel ?p - platform)
        :precondition   (and
                            ;; redundant but useful
                            (at ?vessel ?p)
                            (is-docked ?vessel ?p)
                            (platform-can-refuel ?p)
                            (not (
                                vessel-once-refueled-at-location ?
                                vessel ?p))
                        )
        :effect (and
                        (assign (fuel-level ?vessel) (
                            max-fuel-level))
                        (vessel-once-refueled-at-location ?vessel
                            ?p)
                )
    )

    ;; vessel refuels at port
    (:action refuel-vessel-port
        :parameters (?vessel - vessel ?p - port)
        :precondition   (and
                            ;; redundant but useful
                            (at ?vessel ?p)
                            (is-docked ?vessel ?p)
                            (not (
                                vessel-once-refueled-at-location ?
                                vessel ?p))
                        )
        :effect (and
                        (assign (fuel-level ?vessel) (
                            max-fuel-level))
                        (vessel-once-refueled-at-location ?vessel
                            ?p)
                )
    )

    ;; docks vessel in port or platform
    (:action dock-vessel
        :parameters (?vessel - vessel ?where - logistics_location
            )
        :precondition   (and
                            (at ?vessel ?where)
                            (>= (free-docks ?where) 1)
                            (not (is-docked ?vessel ?where))
                            (not (vessel-once-docked-at-location
                                ?vessel ?where))
                        )
        :effect (and
                    (decrease (free-docks ?where) 1)
```

```
                    (is-docked ?vessel ?where)
                    (vessel-once-docked-at-location ?vessel ?where)
                )
    )

    ;; undocs vessel from port or platform
    (:action undock-vessel
        :parameters (?vessel - vessel ?where - logistics_location)
        :precondition    (and
                        ;; redundant but useful
                        (at ?vessel ?where)
                        (is-docked ?vessel ?where)
                    )
        :effect (and
                    (not (vessel-once-refueled-at-location ?vessel ?where))
                    (increase (free-docks ?where) 1)
                    (not (is-docked ?vessel ?where))
                )
    )
)
```